Automatic Adaptation of Tunable Distributed Applications

by

Fangzhe Chang

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

January 2001

Research Advisors:

Vijay Karamcheti ______________________

Zvi M. Kedem ______________________

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **JAN 2001**	2. REPORT TYPE	3. DATES COVERED **-**
4. TITLE AND SUBTITLE **Automatic Adaptation of Tunable Distributed Applications**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES **The original document contains color images.**		
14. ABSTRACT **see report**		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**159**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

To my family for their support in all these years

Acknowledgment

I am indebted to my family, which have given me strong support in all these years to pursue my own ambition. Special thanks to my advisors: *Zvi M. Kedem* and *Vijay Karamcheti*. It is from Zvi that I learned to appreciate simplicity and to look for interesting problems from ordinary tasks in the computer world. It is Vijay who showed me how to partition problems, write papers, give presentations, and keep extending current work for further goals and development.

I have been working in a very good team. I had enormous help from Ayal Itzkovitz, Arash Baratloo, Tao Zhao, Ninghui Li, Xiaodong Fu, Anatoly Akkerman, Ting-jen Yen, Ee-Chien Chang, Naftali Schwartz, and Hiroshi Ishikawa in my project. I enjoyed discussions and help from all those people: Hseu-Ming Chen, Allen Leung, Karl Holger, Weisong Shi, Mehmet Karaul, Yuanyuan Zhao, Hua Wang, Ilya Lipkind, Anca-Andreea Ivan, Gediminas Adomavicius, Deepak Goyal, Chen Li, Xianghui Duan, Zhe Yang, Fabian Monrose, Peter Piatko, Churngwei Chu, Peter Wyckoff, Niranjan Nilakantan, Madhu Nayakkankuppam, and Shih-Chen Huang. Many thanks to all of them.

This research was sponsored by the Defense Advanced Research Projects Agency under agreement numbers F30602-96-1-0320, F30602-99-1-0157, and N66001-00-1-8920; by the National Science Foundation under CAREER award number CCR–9876128 and grant number CCR-9988176; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copy-right annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, SPAWAR SYSCEN, or the U.S. Government.

Abstract

Current-day applications are written to execute on a wide range of platforms ranging from fast desktop computers to mobile laptops all the way to hand-held PDAs and cellular phones, spanning several orders of magnitude in processing, storage, and communication capabilities. Applications running on these platforms may exhibit diverse and unpredictable performance because of platform heterogeneity and varying resource availability, resulting from competition with other applications or from migration of applications themselves to a different execution environment. However, application users typically expect a desired level of Quality of Service (QoS), either in terms of performance optimization (e.g., shortest completion time) or in terms of predictability (e.g., bounded completion time). This motivates development of *resource-aware* applications, which proactively monitor and control their utilization of the underlying platform, and ensure a desired performance level by adapting themselves to changing resource characteristics.

In this dissertation, we describe an application-independent adaptation framework that simplifies the design of resource-aware applications. This framework eliminates the need for adaptation decisions to be explicitly programmed into applications by relying on three components: (1) a *tunability interface*, which exposes adaptation choices in the form of alternate application configurations while encapsulating core application functionality; (2) a *virtual execution environment*, which emulates application execution under diverse resource availability enabling off-line collection of information about resulting behavior; and (3) *run-time adaptation agents*, which permit monitoring and steering of the application execution. Together, these components permit automatic run-time decisions on *when* to adapt the application behavior by continuously monitoring resource conditions and application progress, and *how* to adapt by dynamically choosing an application configuration most appropriate for the prescribed user preferences.

We evaluate the framework using a distributed image visualization application and a parallel image processing application. The framework permits automatic adaptation to changes of execution environment characteristics such as network bandwidth and image arrival pattern by switching application configurations to satisfy user preferences for output quality and execution timeliness.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Current-day applications are written to execute on a wide range of platforms ranging from fast desktop computers, mobile laptops, all the way to hand-held Personal-Digital-Assistants (PDAs), cellular phones, and PC watches, spanning several orders of magnitude in processing, storage, and communication capabilities, and differing in the system support for application execution. Current desktop machines can have multiple gigahertz (GHz) processors, a gigabytes (GB) of memory, a hundred GB of hard disk, and gigabits-per-second (Gbps) network cards. Laptop machines are typically less powerful due to constraints on their size, weight, and battery life, with a single CPU, less memory, smaller hard disk, and lower bandwidth network connectivity. The power of PDAs is further limited: A Palm V only has a 16 MHz CPU (Motorola 68328) and 2 megabytes (MB) of memory. These lower-end devices could be connected using infra-red, wireless, and bluetooth [32] facilities; thus achieving different rates of data transmission.

With the trend of “write once, run everywhere” software, more and more applications are able to execute on these heterogeneous platforms or even migrate from one platform to another at run time. For instance, Microsoft provides Win32 Application Programming Interface (API) for all its Windows platforms such as Windows NT [55] on desktop machines and or Windows CE [6] on hand-held PDAs (although Windows CE only implements a subset of APIs supported by Windows NT). Applications written using Win32 API can potentially execute on both desktops and PDAs running Microsoft operating systems. With the popularity of virtual machines as execution platforms (e.g., the Java virtual machine (JVM) [47] or the Microsoft .net platform [56]), it is possible for the same application to execute on heterogeneous devices with different OSes. For example, a Java application may be able to run on both Palm Pilot PDAs with PalmOS and desktop machines with the Windows NT operating system, even though the K Virtual Machine (KVM) [59] on the former is only a core subset of a full-featured JVM running on the latter.

However, these applications may exhibit diverse and unpredictable performance due to heterogeneous power of physical platforms in processing, storage, and communication capabilities, as well as to dynamically varying resource availability resulting from competition with other applications or migration of (parts of) applications to a different execution environment [50].

However, such widely varying performance is at odds with the expectations of application users who typically expect a desired level of Quality of Service (QoS). For example, a user viewing a large image over the network may expect that the response time be reasonable. Because this cannot be met on all platforms for a high resolution image, we take advantage of the fact that the user’s expectation are not absolute—there is some flexibility in expressing his expectations: he may prefer a detailed map when using a powerful desktop computer with a high bandwidth network link; however, he may prefer responsiveness at the cost of a lower resolution when using a connected hand-held device with a thin network link.

However, taking advantage of this flexibility in user expectations requires the ability to express user preferences as well as applications' capability to deliver these preferences despite different execution scenarios. More generally, this motivates the design of *resource-aware applications*, which proactively monitor and control its execution and utilization of the underlying platform, can ensure a desired performance level by adapting themselves to changing resource characteristics. For instance, a distributed application conveying a video stream from a server to a client can respond to a reduction in available network bandwidth by compressing the stream or selectively dropping frames. In other words, applications need to be aware of alternate ways to execute as well as the execution environment, and dynamically select an appropriate behavior to adapt to changes of resource availability and satisfy user preferences. These alternate ways to execute could be statically programmed, or dynamically generated and injected into executing applications. In essence, they correspond to different execution paths and provide tradeoff among resource requirements, application performance, and even functionality.

1.2 Application Adaptation

An application may have multiple forms that suit different execution scenarios. In the application source code, these multiple execution forms may take different algorithms to achieve the same functionality, different settings for the same algorithm, or consists of different modules for add-on and more refined computation. At run time, these multiple forms present flexible ways of execution, resulting in different execution path and more importantly different resource requirements and performance levels. The various ways of execution at run time can be viewed as different *configurations* that could be selected based on the changes of resource conditions.

The importance of adapting application behavior to the run-time scenarios such as resource conditions or external events has been understood for a long time. The traditional way to enable adaptation is to explicitly program it into applications. For instance, the TCP protocol responds to network congestion by adaptively deciding an appropriate size of the sliding window and a timeout period for retransmission. Current TCP implementations have been highly optimized for transmitting a large amount of data. However, the drawback with such an approach is that they presuppose all possible patterns of usage. Optimizations based on these assumed usage patterns may not work well for other scenarios. For instance, it has been shown [8] that the default configuration for TCP yields unnecessary delays for short HTTP request/reply traffic (that could be avoided by a specific setting of socket options). Also, TCP resizing protocols have been shown to be poorly suited to high bandwidth transmission in Wide Area Network (WAN) environment [24].

Although adaptation has been programmed into various applications, little support is available for structuring general-purpose resource-aware applications. Several research projects have begun to address this shortcoming [17, 44, 46, 52, 57, 51]; however, most such efforts place a substantial burden on application developers requiring them to provide explicit specification of both *resource-utilization profiles* (which resources are used at which time and in what quantity), and *adaptation behaviors* (how applications react to changes in resource conditions).

More robust application adaptation is possible if programmers only identify what the alternate configurations are, leaving it up to the run-time system to automatically select an appropriate configuration based on execution environments. We focuses on how to enable the development of adaptive applications without these adaptation decisions to be explicitly programmed; thus simplifying the development and supporting more flexible adaptation policies.

We observe that many parallel and distributed applications contain parameters (i.e., variables) that have multiple reasonable (schemes of) settings, such as the value of timeout for retransmission in TCP. The particular setting or scheme adopted are usually optimized for an expected run-time environment. However, in today's platforms of ever-increasing heterogeneity, the adopted form may not work well in all environments; attesting the need for multiple configurations of applications.

We study the *performance benefit* of automatic application adaptation from multiple application configurations in parallel and distributed environments, in particular focusing upon answers to the following questions:

1. what are configurations of an application?
2. how to specify application configurations?
3. how to evaluate the performance of these application configurations?
4. how to monitor the resource availability of the execution environment?
5. how to dynamically select the appropriate application configuration based on changes of resource availability at run time?
6. what are the performance benefits of multiple application configurations?

The answers to these questions allow us to devise mechanisms to expose different configurations of applications and dynamically choose the appropriate configuration to adapt to changes of execution environment such as resource availability, or in a more general form, occurrences of some specific events. These mechanisms in turn can avoid the explicit programming of adaptation decisions into applications.

1.3 Approach

We assume applications of interest already contain flexibility in their implementation, although they may be fixed to one configuration by default. The execution flexibility is controlled by some control “knobs”, different settings of which correspond to different configurations. However, these control knobs are typically implicit in the program. We abstract different configurations out by promoting these implicit parameters to application control structures that can be manipulated from outside of applications. Thus, application adaptation is the switching from one configuration to another configuration by manipulating the explicit control structures, with decisions made and enforced by system components other than applications themselves. However, to make appropriate adaptation decisions, it is necessary to know the performance levels of all the application configurations as well as the current execution conditions, requiring an entire infrastructure collaborating together to delivered user-preferred performance levels. In this dissertation, we construct a framework that exposes configurations of arbitrary applications, evaluates their performance, and dynamically switches among them to adapt to changing execution environments.

1.4 Contributions

This dissertation proposes, from the point of view of resource management, the notion of *application tunability* as an abstraction for multiple application configurations, and studies the benefit of application tunability. It presents a framework for the automatic adaptation of tunable applications, including language annotations that add *tunability interfaces* to traditional applications, a *virtual execution environment* that permits modeling of application behavior under various resource conditions, and *run-time components* that monitor resource availability and automatically adapt applications for a desired level of service. It evaluates this framework with two example applications: a parallel image processing application called Junction Detection [38] and a distributed visualization application called Active Visualization [11].

1.5 Organization

Chapter 2 introduces the project background and the notion of tunability, and discusses related work. Chapter 3 proposes a general framework for enabling automatic adaptation of tunable applications in response to external changes. Chapter 4 studies the tunability interface: the language annotations that expose alternate configurations of applications, enabling run-time monitoring and control of application execution. Chapter 5 presents the design and implementation of a virtual execution environment that simulates various resource conditions, and describes its use for modeling application behavior. Chapter 6 describes the run-time components that monitor execution environment and steer application execution to adapt to changes of resource conditions. Chapter 7 evaluates the benefit of tunable applications, using simulation to show how tunability can help achieving better system-wide resource utilization. Chapter 8 and Chapter 9 present two case studies, using a parallel image processing application and distributed visualization application. Finally, Chapter 10 concludes and discusses future work.

Chapter 2

Background

This chapter states the problem of application adaptation, defines the terms that will be used in the later chapters, and discusses related work.

2.1 Application Structure

With structured programming, applications are usually written in terms *modules*, with some of the modules aggregated into standard libraries (e.g., libraries for data compression). Lately, with the popularity of object-oriented programming, applications have began to be written by composing more general components together, such as in the Java Beans model [21] or the Microsoft COM model [60]. In this dissertation, the term *component* is used to broadly refer to the modules, libraries, and objects that applications are constituted of. These components are linked with control flow mechanisms such as function invocation, message passing, event triggering, and remote procedure call (RPC).

For *distributed* applications, components can be located and execute concurrently on multiple machines, relying on high-level protocols for communication such as IIOP in CORBA [42], ORPC in distributed COM [60], and RMI in Java. The term *application instance* is used to denote the execution of components on a particular machine. As a special case, client-server applications are viewed as distributed with some components running on one machine implementing the server logic and other components running on another machine implementing the client logic.

For *parallel* applications, a single component can execute on multiple processors (or machines) in parallel. These parallel applications, written in a specialized language such as HPF [45] or CC++ [9], exploit support from the underlying system for communication, synchronization, and concurrent execution. Calypso [3] is such a system described in Chapter 9.

Typically, parallel and distributed applications are written with an expected runtime execution environment in mind, including both the explicit expectation such as the operating systems (OSes) and implicit expectation such as the amount of physical memory and the bandwidth of network link. Realization of the former is critical to the correct functioning of applications. For instance, applications written for Microsoft Windows may not execute at all on the Linux operating system (without an intermediate layer such as VMware [61]). Realization of the latter may only affect the application performance. For instance, applications expecting high network bandwidth may not perform optimally under a low bandwidth situation, though they may still be able to execute correctly. In such a scenario, changing of communication protocols may result in better performance.

With the trend that software components are increasingly becoming standardized and their execution environments more heteregeneous, mobile, and dynamic, the performance of applications are more unpredictable and showing a wider range of variance. Nonetheless, application users are demanding a desired level of *Quality of Service (QoS)*, expecting applications to execute in a predictable fashion.

The term *QoS* was first introduced in networking research for guaranteed or differentiated service. It is used in this dissertation to denote performance of applications, with *service* referring to application computation and communication functionality and *quality* referring to performance metrics. QoS permits application users to specify a preferred performance level in terms of the metrics of interest, as well as how performance should be degraded if the preferred level cannot be achieved. For example, when the network bandwidth drops, a user viewing a large image can choose to wait longer or to be satisfied with a lower resolution. QoS provides applications two levels of control. At the first level, applications can adapt to maintain the same level of performance requested by users despite of changes in execution conditions. At the second level, applications can choose how to degrade to a less preferred level. The essence of QoS is that it permits users to have different choices for the same functionality.

2.2 Adaptation for Performance

Since the heterogeneity and dynamic changes of resource availability in application execution environment often result in unpredictable performance, it is necessary for applications to cope with different execution conditions to ensure a specific level of user satisfaction. These conditions could include the capacities of physical resources such as CPU, memory, disk, and network; properties of logical resources such as files and software services; as well as characteristics of input data and user interactions. In an ideal scenario, adaptation for performance allows applications to maintain a prescribed performance level even in presence of changes of execution conditions. More practically however, degradation of performance level might be satisfactory if users can control how performance should be degraded.

For applications composed of software components, adaptation can be achieved using mechanisms belonging to two categories: (1) changing *behavior of components* and (2) changing the *components that make up applications*. The mechanisms taken to shield certain changes in execution environment are typically specific to the applications themselves or the type of the data they process. These mechanisms could be programmed into the applications or installed dynamically in the middleware such as acting as a proxy between application instances. In either cases, the application can be made aware or unaware of the adaptation.

This general problem of adaptation for performance also make it necessary for users to specify their constraints of performance. The latter is usually described by a vector of metrics of which different users may regard the importance in different orders. In essence, these constraints specify policies for adaptation: for instance, an adaptation can only degrade 20% of one particular performance metric within the first 30 seconds; or a metric must have an absolute value smaller than a given figure. In general, enforcing these constraints requires the knowledge of how a particular configuration performs with the current execution condition. This knowledge could be either statically collected, or dynamically measured, or use a hybrid approach. The enforcement could consider only the requirement and performance of the single application, or take into account the competition or collaboration among multiple co-existing applications.

In this work, we restrict our attention to applications with flexibility in configurations statically programmed instead of dynamically injected, to simple interface for specifying user-specific performance constraints, and to the static collection of configuration behavior. It provides an interface so that various enforcement policies could be plugged in. However, it does not focus on how various scheduling algorithms that enforce a particular policy.

2.3 Related Work

Although examples of component-level adaptation (e.g., TCP congestion control) abound in literature, here we focus our attention to application-wide adaptation involving multiple components, which has recently began to attract a great deal of attention.

The **Odyssey** project [51] provides a platform for application-aware adaptation that changes data fidelity of applications (i.e., quality of the output) in response to resource changes (such as that of network bandwidth). It relies on operating system or middleware to monitor resource conditions and requires knowledge of specific data types (e.g., JPEG). The latter enables type-specific definition for fidelity and adaptation mechanisms (e.g., degrading the resolution), using type-specific *warden* components. Inside a warden, various adaptation decisions could be made and actions performed, for instance, prefetching and caching data of a lower fidelity. Odyssey provides an API interface for applications communicating with the warden. In essence, the warden acts as a filter on the path of the data transmission and can take advantage of type-related mechanisms for a particular type of data. Thus, wardens rely only on type-specific knowledge instead of application-specific knowledge.

The **Darwin** project [52] permits a flow-centric application to specify its resource requests in the form of a virtual mesh of nodes (representing desired services) and edges (denoting communication flows). The virtual mesh can be mapped to physical service nodes and links in alternate ways to permit optimization of available resources. It allows application-specific “delegates” to be associated with flows for detecting and handling changes to flow metrics; and thus decides the best mapping of application virtual mesh to physical service nodes and links.

The **Ninja** project [28] intends to build upon and expand the notion of Web-based services by providing *composibility* (the ability to automatically aggregate multiple services together into a single entity), *customizability* (the ability for users to inject code into the system to customize a service’s behavior), and *accessibility* (the ability to access the service from a wide range of devices, including PCs, workstations, cellphones, and PDAs). It proposes to use a *path* composed by strongly-typed *operators* (i.e. software components) to describe the application structure between services and end-users on the Internet. *Active proxies* could be injected into the path to customize the data and access protocols (e.g., translation between HTML and WML) between the end-user devices and the services, thus enabling dynamic service adaptation. Although injecting the intermediaries into the data flow path does not require modification of end-points application instance as in Odyssey, it does require programmers to have knowledge of the protocols and data formats and rely on application usage of these standard protocols and formats.

The **Active Harmony** project [44, 33] presents and exposes application alternatives through an interface based on TCL scripts. It requires applications to specify the resource utilization of each of their execution options as well as the corresponding performance. With this information, the system selects the best option to execute. For instance, it could dynamically select between two different placement options of a database queries based on server load: query-shipping where queries are executed at the server and data-shipping where queries are executed at the client.

The AppLeS project [4, 58] provides application-level mechanisms and resource monitoring tools to enable general applications to adapt to changing resource characteristics. Applications can have customized resource utilization policies and consequently different execution schemes. For instance, applications can decide their own task placement (of application instances to machines) based on execution conditions. AppLeS tools support incorporation of these policies into the execution system for optimized performance.

In **Quality Object (QuO)** framework [64, 53], applications can be explicitly programmed with different levels of performance (i.e., different contract regions) and performance requirements of remote method invocations. It provides QoS Description Language (QDL) and Configuration Setup Language (CSL) components to support the development of distributed applications with QoS requirements. The former associates QoS constraints to function calls and permits remote method invocations to be dispatched to alternate remote objects. The latter permits application execution to switch between different contract regions based on dynamic system conditions.

The **ERDoS** project [17] proposes dynamic application structuring where a service can be implemented with several logical realizations (using a workflow model provided by the project) and applications can be structured in multiple ways. The selection of a particular structure is based on a user-provided benefit function. For instance, users could choose between different encryption algorithms to achieve different level of security. ERDoS provides System Development Tools (SDT) to allow developers to create applications, describe resources, and specify benefit functions.

The **EPIQ** project [57, 36] supports automatically trading off output quality against resource requirements. It views computations as composed of a DAG of tasks where each task can be associated with QoS metrics and resource requirements. Applications could dynamically negotiate a certain performance level (in terms of QoS metrics) for each task and dynamically compose them together to achieve a specific benefit/utility function.

The **Imprecise Computation** model [35, 34] views computation as consisting of two parts: a mandatory part and an optional part. The mandatory part must be executed for a task to meet the minimum quality requirement while the optional part contributes to an improvement in output quality. Therefore, the amount of computation that is actually executed can be dynamically decided according to the resource availability of the execution environment.

All of these projects support the idea of alternate representations of data (e.g., multiple image resolutions) or alternate execution paths of applications. The dynamic selection of the appropriate option constitutes application adaptation. In fact, the idea of alternatives can be found in quite a few systems. For instance, HyperText Markup Language (HTML) [5] provides an `ALT` attribute for image elements that allow browsers to choose whether to use text or images based on processing constraints or user preference. As another example, a Java applet with `main()` method defined could be executed both inside a browser or as a stand-alone application, providing alternate entry points and potentially execution paths suitable for different resource constraints and security constraints.

We observe that most of the above systems involve a fair amount of effort from programmers. In addition to describing what constitutes different configurations of applications, they also require programmers to specify the specific resource conditions in which a given configuration can operate as well as the performance of the configurations under these conditions. They typically integrate explicit adaptation policies into applications by combining these with application-specific schedulers.

In the ideal case, programmers need only specify what are the alternate configurations, with the system automatically making adaptation decisions based upon conditions of the execution environment. This moves the policy enforcement outside of applications themselves, permitting flexible optimization to possibly dynamically changing objectives. It is this perspective that motivates our work described in this dissertation.

2.4 Our Model

We observe that many applications can be executed in one of multiple configurations: either execute the same component with different parameters or execute entirely different components. These multiple configurations present *tradeoffs* and may be suitable for different execution scenarios. We refer to this flexibility as *application tunability*. Note that in this dissertation, we assume the flexibility already exists in applications although the mechanisms for statically or dynamically injecting more flexibility (such as by loading an appropriate substitute library) is also an interesting problem.

Application tunability refers to an application's ability to *trade off resource requirements over several dimensions*, including time, output quality, and resource type. Tunable applications are able to compensate for a lower allocation of resources in a stage of the computation either by requiring a higher allocation in another stage, or by lowering output quality, or by raising demand for resources of a different type. Application tunability provides flexibility to the underlying resource management system, which can now select an application operating point that improves overall system utilization, while still ensuring that an application meets the user predictability requirements.

Although applications can be written to be adaptive in a case by case fashion, it is desirable that adaptation be supported at the system level without much programmer involvement. The notion of application tunability bridges the composition of applications, resource usage, and performance metrics (i.e., output quality); thus enabling automatic application adaptation. We propose a framework that can evaluate different application configurations, and support automatic configuration to ensure applications to achieve a desired level of performance, while minimizing the programmer's involvement in performance-related considerations. With this specific objective in mind, we can further define our view of adaptation and list what questions have to be resolved to achieve this goal.

Refinement of the adaptation problem Given an application in source code form built of components, possibly with alternate components that can substitute those used in the application, a new application could be constructed in the following fashion. The original component can be extended into a container of components with guard expressions specifying which component to select for a particular execution. All the components in one container achieve the same (or similar) functionality and present the same interface for interaction with other parts of the application. The guard expression controls the execution of the container; in the case of the container that holds only one component, different values of its guard expression may correspond to different execution behaviors of the same component. The values of the guard expressions should be dynamically updated so that the application execution is optimized, based on the external execution conditions, including the characteristics of the input data and resource availability of the execution platform. This updating process is denoted as *configuring* or *adapting* the application since it decides which component is executed and in which way. The components executed along the control flow path constitutes an *execution path*.

To realize this model, several problems need to be resolved:

1. *What language constructs are needed to specify the interface of components?* These language constructs must support the comparison of the alternate components and define the application-specific meaning of *optimization of execution*. In addition, all the necessary run-time system for application configuration and adaptation should be generated from these language constructs.

2. *How to evaluate the alternate components and different execution paths?* The evaluation examines which selection of the components performs better under certain external conditions and forms the foundation for application configuration at run time.

3. *How to monitor external conditions and the execution of applications?* The resource conditions and application progress are crucial for run-time adaptation. Typically, adaptation is triggered by a change of external conditions or application internal state.

About problem 1 We use annotations to application source code to allow specification of alternate application configurations. These annotations, collectively referred to as the *tunability interface*, expose different "knobs" to control application execution. Annotations have been in practical use as language extensions for a long time. For example, directives have been used for helping compiler tools (e.g., specifying alignment of data structure) and for parallel processing systems (e.g., for programmers to describe parallel tasks). Similarly, Java Server Pages (JSP) [25] supports annotation of HTML pages using Java code and special tags (e.g., `tag <%!` for class-level declarations) so that execution of the Java annotation combined with HTML can generate dynamic web pages on the fly.

Export of control interface can also be supported by other mechanisms besides source-level annotation. For instance, publicly-declared variables and methods in Java applets are implicitly exported to Javascript code in the same web page, allowing the latter to control the former's execution [63]. Microsoft Component Object Model (COM) [60] allow specification of properties (i.e., variables) and interfaces (i.e., functions) visible for outside usage at the binary level (i.e., in EXE or DLL files). JavaBeans [21] supports a similar idea for the Java programming language. As an enhancement, these models can support exporting multiple interfaces of an component and provide reflection mechanisms to learn about and invoke different interfaces. This ability could be combined with application tunability to export interfaces for selecting different application configurations. Ideally, future applications would all written to export necessary information so that different execution behavior could be easily controlled. However, for applications designed without exposing alternate behavior in mind, source-level annotation or binary-level interception is needed to enable adaptation, which is the approach we pursue in this work.

About problem 2 We use profiling approach to evaluate different configurations and obtain their behavior under various resource conditions. This requires the control of different levels of resource availability to applications. Existing approaches such as [49, 40, 41, 2, 7, 54] permit reservation of resources or fair sharing of resources, which can be used to guarantee applications a certain resource level. However, all these systems require modification of OS kernels and are not designed for simulating different resource conditions. This motivates the design of our own user-level virtual execution environment on COTS operating systems (such as Windows NT or Linux) that simulates resource availability to applications. It provides the basic building block for driver programs that enumerate application configurations and vary resource conditions to obtain application performance behaviors.

About problem 3 We build monitoring tools to estimate resource availability such as that of CPU and network; we also structure applications in a flexible way to allow external events to be sent to them. The latter permits plugin of other monitoring tools into the our framework.

Many projects have studied the monitoring of system resources. For instance, the Remos system [48, 31] presents a structure for collecting resource information and a set of APIs for applications to query resource information. Apertos [39] proposes building adaptive operating systems using the reflection mechanism. Meta-level objects (reflectors) are used to hold the information about system components and act as an interface to manipulate the components' execution environment.

Our framework could plug in existing monitoring systems as external agents to attain environmental information. For studying our example applications, we have developed our own monitoring agents that keep track of CPU and network resource availability or other external events.

Based on an understanding of these three problems, we can build a framework that avoids explicit programming of adaptation decisions into adaptive applications. Our framework differs from the existing systems principally in the division of responsibility between application developers and the execution system. Application developers are required only to expose the adaptation structure of applications using the tunability interface. The execution system takes responsibility for obtaining application behavior profiles using a virtual execution environment, and incorporating these profiles into adaptation decisions at run time. The tunability interface insulates the resource scheduler from application-specific knowledge. As far as we know, our strategy is unique in its use of a virtual execution environment for automatically modeling application performance under diverse resource conditions.

Chapter 3

An Adaptation Framework for Tunable Applications

Many applications exhibit "tunability" opportunity: the existence of multiple ways of execution. One of these configurations may deliver the best performance in a specific execution environment; however another one may perform better in a different environment. We can build a framework to expose, parameterize, and control the selection of application configurations based on execution conditions. This chapter elaborates the notion of application tunability and describes the application adaptation framework. The framework was originally proposed and refined in [13, 14].

3.1 Application Tunability

We observe that many applications implicitly contain parameters (i.e., variables) that control the way applications execute. These parameters typically give different settings for application component, turn on or off the execution of a component, or control the selection of alternate components in the execution path. In software development practice, typically a large amount of alternate components are developed to achieve the same functionality with different algorithms (e.g., various sorting algorithms and compression methods). These alternate components may be statically included by application developers into applications as an effort to explore the best setting for an expected execution scenario, or dynamically included by loading a shared library (e.g., DLL) or class file, or by injecting intermediaries between application instances. In our model, we assume that this flexibility already exists in applications; and we focus on how to expose, parameterize, and control it. This assumption is satisfied by a large number of applications. For instance, a component for retrieving a multi-resolution image may have a local variable controlling the resolution level; a lossy compression algorithms may have a parameter controlling fidelity of the data; a parallel application may have a variable reflecting the number of processors it is using and partition its data set accordingly; a web browser may have a variable to turn on or off the use of a proxy; a fault-tolerance algorithm may have a parameter deciding the number of replicas to use; a database server may have a variable controlling alternate algorithms (i.e. components) to use to build indexes on a particular database. In execution, the different settings for these parameters may require different amount of resources and deliver different levels of performance.

Tunability refers to an application's ability to trade off resource requirements over several dimensions, including time, quality, and resource type, while still producing an output of adequate quality. Application tunability is a characteristic of many parallel and distributed computations. The key attribute unifying all tunable computations is the availability of *alternate application configurations*, each corresponding to a different path of execution and resource requirement profile. The differences in the resource utilization profiles of the alternate configurations can be characterized as tradeoffs along several dimensions: (i) *time*, (ii) *resource types*, and (iii) *output quality*.

Trading off resource requirements over time means that a large allocation of resources in one stage of the computation's lifetime can compensate for a small allocation in another stage, or vice versa. For example, an artifact recognition application may first sample different portions of the image to decide on interesting regions, and then run a resource-intensive algorithm on these regions; spending more resources on the sampling step reduces the work that will need to be performed in the analysis step.

Trading off resource requirements over resource types means that a large allocation of a particular type of resource can compensate for a small allocation of another type of resource, either in the same or a different stage of the computation's lifetime. For example, a multimedia data transmission application may send data either in compressed format or in its original form. Choosing a compressed format would save network bandwidth at the expense of more computational power required for on-the-fly compression and decompression. The application can thus trade off computational resources versus network resources by deciding whether or not to compress the data before injecting it into the network.

Trading off resource requirements over output quality means that applications can compensate for a reduced resource allocation by varying the quality of output, while still operating in an acceptable range. For example, in several scientific computation applications where approximate results are acceptable, investing additional compute resources beyond an acceptable point will improve the accuracy of the result.

Along these tradeoff dimensions, application tunability provides flexibility to the underlying resource management system, which can now use the choice in resource allocation profiles to increase the number of applications that can be admitted into the system, as well as for ensuring that the user preferences for performance levels are satisfied.

Example applications In this dissertation, we use two example applications to study how tunability enables adaptation to changes of external conditions, such as network bandwidth or the overall resource requirements for system resources. Details are described in Chapter 8 for the *Active Visualization* application and in Chapter 9 for the *Junction Detection* application.

Active Visualization is a client-server image visualization application that allows users to interactively view large images stored at the server side. It uses multi-resolution and progressive transmission techniques. For each user interaction, the client sends a request for data of certain area (controlled in the program by a parameter named `f`) and of a certain resolution (controlled by a variable named `l`). The server retrieves the data of the requested resolution and compresses it with some compression method (indicated by a variable named `c`), and sends the data back to the client. This process continues until the entire image is transmitted from the server to the client. The choice of these parameters corresponds to different configurations of this application, and leads to different resource requirements. For instance, the choice of different compression methods may trade off the requirements of CPU and network resources because they may achieve different compression ratio with different amount of computation. For this application, tunability allows flexibility in the way the application is executed and the way user preference (such as that for a high image resolution) is expressed.

Junction Detection is a parallel image processing application that detects pixels where color or intensity changes abruptly (i.e., junctions). It has three steps: the first step samples the input image with some granularity to find interesting points; the second step draws a convex hull around those interesting points to define interesting regions; and the third step performs an intensive computation to find all of the junctions inside interesting regions. Two of its parameters are interesting to us: the sampling granularity and the number of processors allocated to the parallel computation. For the first parameter, a finer sampling granularity may reduce the size of the interesting region and thus the amount of computation in the third step; however, at the cost of more computation to perform the sampling itself. The amount of increment or decrement in computation depends on the complexity of algorithms used in these two steps and is moreover influenced by the characteristics of the input data. For the second parameter, more processors allocated to the application will lead to a shorter time to process the image. However, with more processors allocated, the application may not be able to use them all efficiently; in other words, the speedup function may not be linear. The choice of this parameter allows the tradeoff of the performance of this single application against the overall system resource utilization. For this application, tunability allows optimization of application performance based on external events, such as the characteristics of the input data and overall requests for system resources.

3.2 Adaptation Framework

Application tunability can help to achieve application adaptation: some of the multiple configurations may better match different execution conditions because of the tradeoffs in the resource requirements of these configurations. A resource management architecture, which is aware of the multiple configurations, can exploit the differences among their resource utilization profiles to select a configuration that best matches the characteristics of available system resources. This requires exposing these configurations to the outside system, parameterizing them in terms of their matching resource conditions and corresponding performance level, and devising appropriate components and interfaces to monitor resource availability and dynamically selecting the best configurations to satisfy user preference.

Given an application consisting of components in source code and some alternate components, our approach first annotates the application source code with specialized language constructs called the *tunability interface,* exposing the alternate configurations (see Figure 3.1). A preprocessor processes the *annotations* and generates a tunable version of the application. The critical feature of this tunable application is the existence of multiple execution paths, selecting among which can happen dynamically at run time.

To model the behavior of the various configurations and obtain what kind of execution conditions they suit for, these configurations are evaluated on a *virtual execution environment* with different resource conditions, obtaining a *performance database* that maps application configurations and resource conditions to the performance metrics.

At run time, user preferences are given in terms of *QoS constraints*. Dynamic resource conditions are monitored by a *monitoring agent* plugged in by the preprocessor. A *resource scheduler* obtains the resource information from the monitor, and computes the most appropriate configuration for the application. It controls the application's execution by sending control messages to its *steering agent* generated by the preprocessor, which in turn sets up the application to execute the particular configuration.

During execution of the application, the monitoring agent keeps track of both the

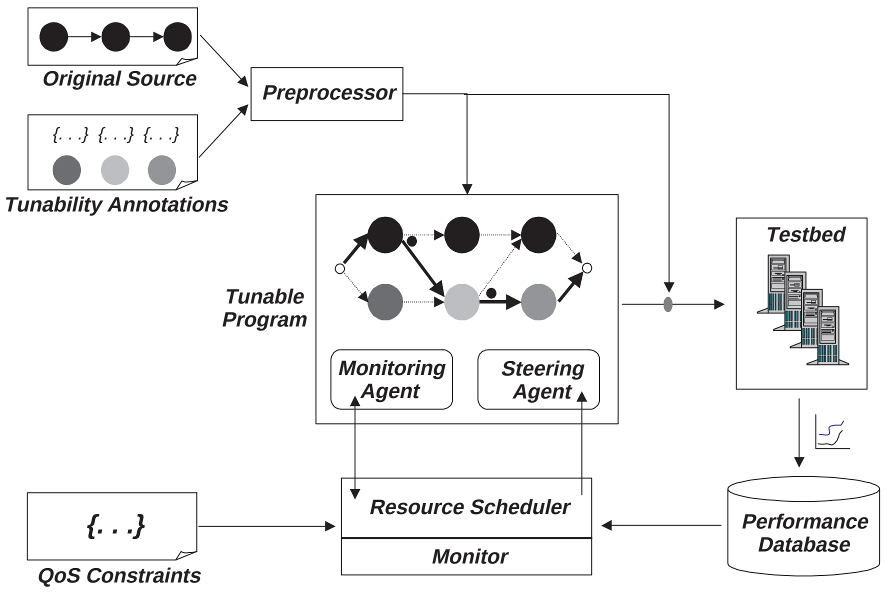


Figure 3.1: Application configuration and adaptation framework.

resource conditions and application progress, triggering a performance exception when either of them drops below a threshold. This exception signals the scheduler to recompute the most appropriate configurations the application must adapt to in order to best satisfy the user preferred performance level.

The following chapters will describe the various components in the adaptation framework. Chapter 4 discusses the tunability interface used for annotating the original applications. Chapter 5 introduces a novel tool providing the basis for the virtual execution environment for profiling application configurations. Chapter 6 describes various runtime components, including the monitoring agent, the steering agent, and the resource scheduler. The evaluation of this framework using our example applications are described in Chapter 8 and Chapter 9, respectively.

Chapter 4

Tunability Interface

Although many applications are developed with specific execution scenarios in mind, they could be ported to suit other conditions by slightly changing their execution behaviors; for example, only computing an imprecise output when resources are not abundant. This flexibility allows static applications to be promoted to tunable applications. This promotion in general is performed in a stage *independent of application development*; nonetheless, it requires *application-specific knowledge*, including (1) identification of application configurations, which form the basis of adaptation, (2) monitoring of the execution environment, which provides context of the execution and events that applications need adapt to,(3) measurement of performance metrics of interest, which gives performance levels of the selected configurations in a profiling stage and performance feedback at run time, and (4) ultimately selection of appropriate configurations based on monitored execution scenarios, which is achieved in an application-specific fashion. Although each of these steps could be done manually on a per application basis, they are better controlled at the system level without extensive programmer involvement. Realizing this objective requires some application-independent way of enumerating what the different configurations are as well as mechanisms for effecting transitions among the various alternatives. We meet these requirements by abstracting the structure of general tunable applications with language-level constructs, called the application *tunability interface* first introduced in [15, 16]. The tunability interface provides language support to express availability of multiple execution paths for applications, and provides means by which application progress can be monitored and influenced (by switching to a different path), thereby alleviating the programmers' burden of writing tunable applications.

With these constructs, we identify

1. the implicit parameters and local variables that control the behavior of applications and promote them into application control structures that is exposed to the outside systems;
2. the implicit parameters and local variables that reflect application performance, with possible introduction of new variables that measures other performance metrics and expressions that specify how these new performance metrics should be evaluated;
3. the software components whose behavior can be changed as nodes along the execution path;
4. the points at which adaptation can happen, either inside a component or between component boundaries;
5. application-specific actions to be taken as part of adaptation process;
6. the resources and external events this application need adapt to. The application needs to export an standard interface to obtain the notification of these events while the system need to keep monitoring of them.

The language constructs for exporting these information could be divided into two categories: application structuring constructs and application monitoring and control constructs. Section 4.1 and Section 4.2 discuss these two kinds of constructs respectively. Section 4.3 assumes that application source code is available and shows a concrete implementation of these constructs at source-code level. Section 4.4 shows how to cope with binary applications where source code is unavailable, using code interception and injection techniques so that new source code could be defined for binary functions.

4.1 Application Structuring Constructs

As described in Chapter 1 and Chapter 2, applications are assumed to be built up from multiple components linked together using standard control-flow mechanisms, with application configurations result in different behaviors for these components. These configurations are exported using the tunability interface. Two kinds of constructs are used to specify the different configurations of applications:

Control parameters provide the "knobs" by which component behavior can be influenced. Setting different values to the control parameters results in the component following a different execution path. These parameters are usually local variables inside a program module; but are promoted as part of the tunability interface by giving an external name and listing in the *control_param* construct.

The local variables and control parameters exchange their values when the application execution enters or leaves the corresponding module. The value of control parameters could also be changed through an external notification (i.e., control messages) from a system scheduler or another application instance. The change would be reflected in local variables at adaptation points specified by other constructs.

For parameters that only accept a certain values, the constraints such as a value range could be specified. The system will generate filters when setting of new values to prevent invalid executions.

Tunable components describes tunability of at component level. Each tunable component is viewed as a container of modules of fixed behaviors (as described in Chapter 2), with guard expressions selecting a particular behavior for execution. Tunable components are linked together by inter-components control flow to set up the alternate execution paths. The abstract model of a tunable application is that of a family of DAGs built up from individual tunable components.

Each tunable component is specified by a *component* construct, which describes the control parameters affecting module execution, the environment (in terms of events and resources) that are of interest to the component, and application-specific QoS metrics that denote component output quality (see Section 4.2 for detail). As a result, a tunable component is parameterized: its execution is determined by the control parameters and its performance reflected by the quality metrics.

To identify different configurations of tunable components in the execution path, tunable components can have names in which control parameters can be included. The control parameters in the component name are evaluated as name-value pairs when the component is instantiated at run time, serving as a handle for referring to a specific configuration of the component.

A component is typically programmed in the form of a function. Correspondingly, specifying component constructs can be either inside the function definitions or wrap around function calls (where function definition is not available). We insert special inline code at the boundary of components. When entering or leaving components, various actions could be introduced by the inline code, including recording the configuration and performance levels, query the resource conditions, or trigger adaptation. The body of a component is left unspecified, and can include other constructs (e.g. control parameters), nested components, and original source code of the software module for this component.

To include or exclude the entire component in the execution path, we introduce guard expressions for each component. The guard expression are valid expressions in the source language, however, contains only operations on control parameters specified within this component.

Note that although it is the different execution paths and different behavior of components on execution paths that provide flexibility in application execution, the tunability interface relies on that this information are properly reflected in the different settings of control parameters. As a result, each specific setting of control parameters corresponds to one meaningful execution path. In this sense, the control parameter constructs describe the different configurations of applications while components constructs act as containers of software modules and as units for controlling application execution.

4.2 Application Monitoring and Control Constructs

Execution environment specifies the system entities (hosts and network links) on which application components execute, as well as the external events the application is interested in. Each system entity in turn encapsulates several resources that affect application behavior. For instance, a host is characterized by its CPU, memory, and network resources. The monitoring agents are designed to monitor the conditions of these resources and propagate them to local and remote application instances. The monitoring agent could either act as part of the application or reside on each host outside the application. The objective of execution environment specification is to provide a standard interface between the monitoring agents, applications, and other parts of system (such as schedulers) to exchange information about execution context. Besides the physical resources, the application may be interested in the occurrence of other events, such as a change in the number of active users in the system. These events are also declared as part of execution environment to support event notifications. One limitation of this approach is that these events should be specifiable with simple data types such as an integer or a string. It is assumed that some external agents monitor and notify the tunable application about the occurrence of these events.

The notification of changes of environment parameters does not directly affect application execution. Instead, they are treated as factors used to decide appropriate settings of control parameters. As the result of a notification, the control parameters (and correspondingly the execution path) may or may not be changed, depending on the decision of resource schedulers (see Chapter 6 for details).

QoS metrics and evaluation expressions specify the component metrics of interest as well as application-provided means of computing them. These QoS metrics (declared as *QoS_param*) are viewed as measurements of application performance. In general, performance metrics are application-dependent although some may be applicable to all applications (such as the total time for processing the input data). Consequently, the tunability interface should not need to know the semantics of these metrics. However, it does require that they are comparable with minimum or maximum values and are simple data types, in order to optimize values of these metrics.

For many applications, performance metrics are mapped to local variables or computable from values of local variables. For the latter case, the tunability interface provides *QoS_monitor* constructs to compute these metrics. Typically, QoS_monitor constructs are annotations nested within the *component* constructs, evaluating various quality metrics associated with this component.

Transition functions specify the points at which adaptation can happen and encapsulate application-specific actions required upon a change in the component configuration (e.g., updates to local variables and/or sending control messages to other components). Compared to component constructs that allow adaptation at the entrance of a module, transition functions enable adaptation inside a component.

Transition functions take two sets of control parameters as parameters, identifying the current configuration and the configuration suggested by schedulers based on the current conditions of execution environment. The transition function code could examine these two sets and decide how to proceed; possibly overriding the system's decision. At the end of executing this code, the control parameters obtain the setting of the new configuration. Usually adaptation decisions are made automatically by the system (outside of applications), transition functions provides ways by which application could constrain this decision.

4.3 Source-level Implementation of Tunability Interface

One way of realizing the tunability interface is as annotations to application source code. These annotations can then be translated into the same language as the application and compiled together to generate tunable versions of applications. These annotation constructs could be macro definitions in the source language, or elements and attributes in Markup languages. In implementation, we use one of the Markup languages—eXtensible Markup Language (XML) as a generic language for expressing tunability interface. The annotations permit language-specific code in special places such as guard expressions or transition functions in component constructs. Currently, these annotations only translate to C/C++, using a preprocessor implemented on top of a XML parser. The control parameters, QoS metrics, and execution environment definitions are converted into data structures in the source language (e.g., structure/class declarations in C/C++), accessible to other inline annotation constructs (e.g., QoS_monitor). Different execution paths are achieved by expanding the tunable component constructs into conditional statements involving the associated guard expressions. Changes in component configurations are detected by inline checking code. In addition to the tunable versions of each component, the preprocessor also generates *monitoring* and *steering* agents that monitor resource availability and control application execution respectively (see Chapter 6 for details). These agents are aware of the definition of control parameters and QoS metrics, and can communicate with each other via control messages that specifies the parameter names and new values.

The following explains how these constructs look like in XML and their corresponding counterparts in C/C++ code.

Control parameters refer to local variables inside a component which are exposed by giving external names and listing in the *control_param* construct as control "knob". For example,

```
<control_param>
```

```
   <param type="int" localname="c" externname="compress" />
</control_param>
```

identifies a local variable `c` of type `int` and promotes it as a control parameter `compress`. As a result, the control parameter structure would have `compress` defined,

```
typedef struct {
    ...
    int compress;
    ...
} control_param;
```

A global variable will be declared to represent the current setting of all of the control parameters; auxiliary functions are also defined, for instance, to support the copy, serialization, and deserialization of the structure.

Components are wrappers of tunable modules with guard expressions that could be turned on or off. The following construct identifies function `compress()` as a tunable module:

```
void compress(...) {
<component name="module{compress}" >
   <condition> expr </condition>
    other constructs ...
    // original source code
</component>
}
```

describes a component with guard expression `expr` and a name comprising of characters and a control parameter `compress`. Inside component constructs, other constructs could be defined such as control parameters, execution environment, QoS metrics, and nested components. As a result, this construct will be translated into

```
void compress(...) {
    if (expr) {
        begin_component("module{compress}");
         code for processing other constructs
         // original source code
        end_component("module{compress}");
    }
}
```

The function `begin_component()` and `end_component()` is defined to record the values of control parameters, resource parameters, and QoS metrics, and to trigger adaptation if necessary. The latter will give the control parameters proper values for executing this module.

Execution environment specifies the context of application execution. We identify a host where an application instance executes on by a name (e.g. name of the application instance), IP address and TCP port the instance listens to for external notifications. The host is characterized by the constituent resources such as CPU, memory, and network resources that available to the instance on this host.

The following construct specifies a host `client` with address `localhost` in the execution environment, listening to port `vSys.port+1` for external notifications. One of its resource parameters is the network condition (as specified by `net`). In addition, it also listens to the notification of an event named `users` whose meaning is unknown to the application (for instance, it might specify the number of users on the local host).

```
<resource_param>
```

```
    <host name="client" address="localhost" port="vSys.port+1" />
    <hostresc type="double" name="net" host="client" />
    <event type="int" name="users" />
</resource_param>
```

These declarations are translated into structure definitions that hold the value for these parameters and the corresponding code for listening to notifications. For CPU, memory and network resources, the corresponding monitoring agents are generated (see chapter 6 for details).

QoS metrics define performance metrics of interest to application components. For instance,

```
<QoS_param>
   <qos type="int" externname="resolution" localname="l"  dir="inc" />
   <qos type="double" externname="response_time" dir="dec" />
</QoS_param>
```

defines two performance metrics: `resolution` maps to the local variable `l`, with a larger value indicating a better performance (as specified by `inc`); and `response_time` would be evaluated in QoS_monitor constructs, with a smaller value indicating a better performance (as specified by `dec`).

This construct translates to corresponding definitions in global data structure (as for control parameters) and synchronization code to map between the local variables and the data structure. As a result, the value of `resolution` synchronizes with the local variable `l` at the end of the component execution. However, additional, *QoS_monitor* constructs (not shown) are required to set values for `response_time`. The code in these constructs are directly embedded into application source code; synchronization code is also generated before and after the QoS_monitor constructs.

Transition functions specify tuning points inside a component as well as application-specific actions needed when switching from one configuration to another. The actions are code written in the same source language as the rest of the application. For instance, the construct

```
<transition from="oldctl" to="newctl">
    application-specific code ...
</transition>
```

defines a transition function with two sets of control parameters *oldctl* and *newctl*, where the *newctl* identifies the configuration suggested by the system based on the current resource conditions.

In addition to processing the constructs described above, the preprocessor also generates file templates for application profiles and user preferences for application execution. These serve as input to both the virtual execution environment (see Chapter 5 for details) and the external resource schedulers. These configuration files identify the application components, and their control parameters, execution environments, and QoS metrics. Availability of this information allows an external source, either a driver program or a system-level scheduler, to flexibly interact with a running application for performing a variety of tasks ranging from passive measurement (e.g., recording the performance achieved by a particular configuration under a given resource condition) to more active control (e.g., changing the behavior of application components).

4.4 Binary-level Realization of Tunability Interface

So far in this chapter we have shown how to annotate application source code to export its tunability interface. However, we observe that some applications have sufficient information available at the binary level to support adaptation. For these applications, we do not need to have access to their source code, or even recompile and relink them to get tunable versions.

The basic approach for realizing the tunability interface at binary level is to intercept application function calls at run time and control their behavior by changing function arguments or return values. A few function interception tools are available on popular operating systems such as Windows NT and Linux. For instance, the standard binary format on Windows NT permits finding the function address for a particular symbol (i.e., function name). Windows NT also allows a process to access the memory image of another and to inject a thread into the latter. Together, these allow rewriting the binary code of that function. In the original binary code of the function, the first several instructions could be changed to jump to the new function definition that are dynamically injected into application address space (by loading the corresponding DLLs). This is how Detours [37] works, a tool we use to intercept and reinterpret function calls. Similar interception is also possible on other operating systems (e.g., Linux), and for Java applications (in Byte Code format by simply reloading class definitions).

Binary realization of the tunability interface assumes some application-specific knowledge besides the binary code itself. Similar to the source-level interface, we also need to identify control parameters, tunable components, QoS metrics, execution environment parameters, and transition functions. Here, the control parameters can only map to formal parameters of the functions, instead of any local variables. In addition, we need to decide which function calls to intercept to inject these constructs into applications. For instance, if a transition function needs to be inserted into a particular point of a component, we require a function invocation at that point. We redefine that function and insert the calling of transition functions in the beginning of the new function.

As the result of the interception, a new library is authored and annotated using the tunability interface. It can be preprocessed using a similar parser as that at the source-code level to generate a shared library (e.g., DLL) that is to be loaded into application at run time. This library contains all the necessary code for monitoring and adapting application execution. As a prototype, we injected a DLL into Active Visualization application, intercepting and redefining some C/C++ functions. For instance, the entry point `WinMain()` is intercepted to initialize the tunability data structures. The prototype which combines the original binary with the injected code achieves the same adaptation as allowed by the source-level annotation as what we report in Chapter 8.

Besides exposing tunability of an application, this approach also allows injection of alternate components into applications that do not originally possess them. For instance, if an application supports a single compress method LZW, using the following prototype,

```
void compress(char indata [], int insize,
              char outdata[], int outsize);
```

We could rename this implementation to be `original_compress` and redefine it to introduce another compression method Bzip2 and control the selection based on a control parameter `compress`, as following,

```
void compress(char indata [], int insize,
              char outdata[], int outsize) {
    int c = control.compress;
    if (c) Bzip2(indata, insize, outdata, outsize);
    else original_compress(indata, insize, outdata, outsize);
}
```

With the new definition, the application can use alternate compress methods chosen by a resource scheduler to optimize its performance.

To generalize this approach, we need language constructs to specify function interception and combine with the source-level annotation constructs. In fact, deciding which function to intercept requires further information, such as a brief control flow graph to specify the relation of function invocations and structure of applications. Identifying them inherently requires human effort. However, since the tunability interface is independent of applications, some tools can be built to facilitate the generation of adaptation code.

Compared with source-level annotation, binary-level realization is limited in that application binaries have less information and provides fewer mechanisms that can be taken advantage of. First, binary-level realization only allows code modification at the granularity of function calls while annotation at source code level permits changes to arbitrary code segments. Second, control parameters and QoS metrics can only map to function arguments while they can map to arbitrary local parameters at source level. Third, type information is most likely unavailable and thus certain operations are impossible. For example, without debugging information present in the executable, it is difficult to determine the type of `this` pointer of a C++ member function and obtain access to its member variables because this information is in general unavailable. Of course, the availability of advanced language features such as reflection in Java and other object-oriented languages somewhat alleviates this difficulty.

4.5 Summary

Different application configurations can be exposed through the tunability interface, which specifies components whose behavior can be controlled, control parameters that decide the execution path, QoS metrics that reflect application performance, and transition functions that implement application-specific actions when switching between application configurations. These specifications provide sufficient information to generate run-time components that permit application adaptation. We showed how to realize this interface at the source-code level using XML-based annotations and at binary level using a prototype based on function interception techniques for shrink-wrapped applications binaries.

Chapter 5

Virtual Execution Environment

In the model described in Chapter 2, application adaptation is the selection of an appropriate configuration among various configurations based on resource conditions and user requirements. Before this can happen, we need to be able to model the behavior of configurations and resource conditions they are most appropriate for. This chapter describes a novel profiling approach, relying on the construction of a virtual execution environment to control various resource conditions available to applications.

Explicit specifications of application tunability enables development of a model of behavior for each of application configurations, expressed as a mapping from control parameters and resource conditions (application input can be viewed as an additional parameter in the execution context) to application-specific quality metrics. The existing approaches for obtaining this mapping can be divided into two categories: analytical and profiling-based approach.

Analytical approaches attempt to model application performance by analyzing the complexity of algorithms it uses and come up with a formula that predicts application behavior on a certain system, under a certain set of resource conditions. However, the difficulty with this approach lies in the complicated nature of both current-day execution platforms and large-scale applications. Current physical systems contain several levels of memory hierarchy from on-chip caches to disk storage or even network file system, use aggressive pipelining and out-of-order execution mechanisms, provide devices with varying characteristics of devices such as network cards, and require complex interactions with operating systems. At the same time, current applications are built with complicated control flow and data flow, even with remote function calls to other physical systems or with multiple instances running at the same time with varying degree of synchronization among them. Both of these factors significantly restrict the general applicability of analytical approaches.

On the other hand, profiling-based approaches treat applications as black boxes, relying upon measurement of their executions to build performance models. However, a shortcoming of this approach is that obtaining a full performance model of the application requires measurements across a broad range of resource conditions. In principle it is possible to obtain these profiles by executing each configuration in different physical distributed environments corresponding to every possible run-time resource availability situation. However, practical considerations rule against this approach because of the difficulty of configuring such a large variety of distributed systems.

Instead, we obtain application profiles using a technique for creating *virtual execution environments* [12]. These environments run on top of a static distributed system, and can be configured to accurately emulate a variety of resource availability scenarios. Our implementation of these environments relies on standard mechanisms available in most current-day operating systems and a novel technique called API interception, and supports the constrained execution of unmodified applications.

This chapter first describes the construction of the virtual execution environment in Section 5.1 and studies its use for profiling application behavior in Section 5.2.

5.1 Construction of Virtual Execution Environment

We implement a virtual execution environment by effectively creating a “sandbox” around an application. This sandbox constrains application utilization of system resources such as the CPU, memory, disk, and network.

Existing approaches for enforcing qualitative and quantitative restrictions on resource usage rely on kernel support [40, 49], binary modification [62], or active interception of applications’ interactions with operating systems (OSes) [1, 23, 27]. The kernel approaches are general-purpose but require extensive modifications to OS structure, limiting their applicability for expressing flexible resource control policies. The remainder of the approaches rely on deciding for each application interaction with the underlying system whether or not to permit this interaction to proceed; consequently, they provide qualitative restrictions (such as whether or not a file-reading operation should be allowed), but are unable to handle most kinds of quantitative restrictions, particularly since usage of some resources (e.g., the CPU) does not require explicit application requests.

We devise a user-level sandboxing approach for enforcing quantitative restrictions on application’s resource usage. Our approach instruments (using tools such as Mediating Connectors [1] and Detours [37]) application binaries, actively *monitors* the application’s interactions with the underlying system, and proactively *controlling* them to enforce the desired behavior. Our strategy recognizes that application access to system resources can be modeled as a sequence of requests spread out over time. These requests can be either implicit such as for a physical memory page, or explicit such as for disk access (however, those disk operations incurred by paging are also implicit). This observation provides two alternatives for constraining resource utilization over a time window: either control the resources available to the application at the point of the request or control the time interval between resource requests. In both cases and for all kinds of resources, the specific control is influenced by the extent to which the application has exceeded or fallen behind a *progress metric*. The latter represents an estimate of the resource consumption of applications.

For this approach, the primary challenge lies in accurately estimating the progress metric and effecting necessary control on resource requests with acceptable overhead. It might appear that appropriate monitoring and control would require extensive kernel involvement, restricting their applicability. Fortunately, most modern OSes provide a core set of user-level mechanisms that can be used to construct the required support. Presence of *fine-grained timers* and *monitoring infrastructures* such as the Windows NT Performance Counters and the UNIX `/proc` filesystem provides needed information for building accurate progress models. Similarly, fine-grained control can be effected using standard OS mechanisms such as *debugger processes*, *priority-based scheduling*, and *page-based memory protection*.

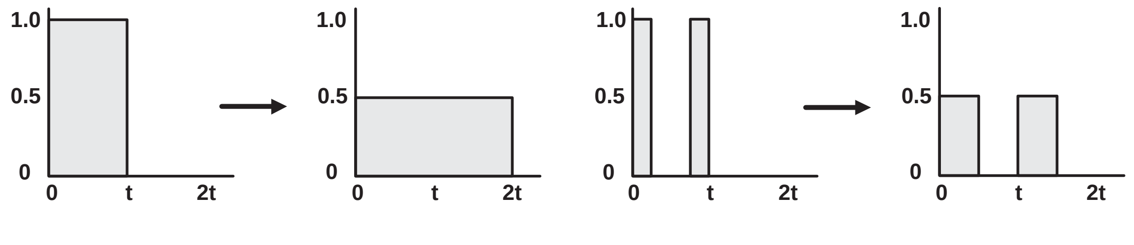
5.1.1 General Strategy

The goal of controlling resource consumption can be two-fold: to simply prevent an application from overusing system resources and starving other applications, or to provide a soft guarantee and weighted fair sharing of resources to the controlled applications. The latter goal can create, for each application, a virtual execution environment that simulates a physical machine with the prescribed resource limitations. However, meeting this goal requires that extra resources cannot be given to the constrained application even if available. In the following, we describe how our strategy can be used to control application consumption of three representative resources: CPU, memory, and network.

CPU resources

For CPU resources, the quantitative restriction is to ensure that the application receives a stable, predictable processor share. From the application's perspective, it should appear as if it were executing on a virtual processor of the equivalent speed.

Constraining CPU usage of an application utilizes the general strategy described earlier. The application is sandboxed using a monitor process that either starts the application or attaches to it at run time. The monitor process periodically samples the underlying performance monitoring infrastructure to estimate a progress metric. In this case, progress can be defined as the portion of its CPU requirement that has been satisfied over a period of time. This metric can be calculated as the ratio of the allocated CPU time to the total time this application has been ready for execution in this period. However, although most OSes provide the former information, they do not yield much information on the latter. This is because few OS monitoring infrastructures distinguish (in what gets recorded) between time periods where the process is waiting for a system event and where it is ready waiting for another process to yield CPU. To model the virtual processor behavior of an application with wait times (see Figure 5.1 for a depiction of the desired behavior), we use a heuristic to estimate the total time the application is in a wait state. The heuristic periodically checks the process state, and assumes that the process has been in the same state for the entire time since the previous check.



(a) (b)

Figure 5.1: Desired effects on application execution time (x axis) under a resource-constrained sandbox that limits CPU share (y axis) to 50% when the application contains (a) no wait states, and (b) wait states. In the latter case, the sandbox should only cause the ready periods to get stretched out.

The actual CPU share allocated to the application is controlled by periodically determining whether the granted CPU share exceeds or falls behind the desired threshold. The guiding principle is that if other applications take up excessive CPU at the expense of the sandboxed application, the monitor compensates by giving the application a higher share of the CPU than what has been requested. However, if the application's CPU usage exceeds the prescribed processor share, the monitor would reduce its CPU quantum for a while, until the average utilization drops down to the requested level. While the application is waiting for a system event (e.g., arrival of a network message), it is waiting for resources other than the CPU. Consequently, the time in a waiting state is not included in estimating the CPU share and the application would not get compensated for being in a wait state. For this scheme to be effective, the lifetime of the application needs to be larger than the period between sampling points where the progress metric is recomputed.

Memory resources

The quantitative restriction of interest here is the amount of physical memory an application can use. The sandbox would ensure that physical memory allocated to the application does not exceed a prescribed threshold. Monitoring the amount of physical memory allocated to an application is straightforward. The monitoring infrastructure on all modern OSes provides this information in the form of the process working set (resident set) size. The progress metric is the application's peak working set size over a period. No control is necessary when the progress is less than the threshold.

However, it is more involved to control the application behavior in case the OS allocates more physical pages than the threshold. The problem is that these resources are allocated implicitly subjecting to the OS memory management policies. The basic idea is to have the monitor act as a user-level pager on top of the OS-level pager, relying on an OS-specific protocol for voluntarily relinquishing the surplus physical memory pages allocated to the application (see Figure 5.2). Also, unlike the CPU case where periodic monitoring and control of application progress is required, here the monitoring and control can adapt itself to application behavior. The latter is required only if the application physical memory usage exceeds the prescribed threshold, which in turn can be detected by exploiting OS support for user-level protection fault handlers.

Network resources

For network resources, the quantitative restriction refers to the sending or receiving bandwidth available to the application on its network connections. Unlike CPU and memory resources, application usage of network resources involves an explicit API request. This permits the monitoring code injected into the application to keep track of

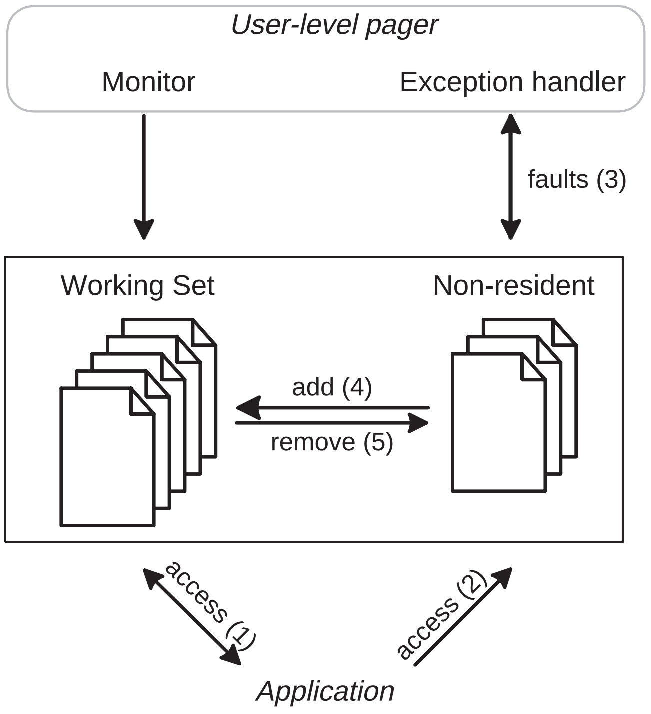


Figure 5.2: A user-level strategy for controlling application physical memory usage. The application has normal access to pages in its working set (1). When it accesses a non-resident page (2), a page fault is triggered (3). The user-level pager adds this page to the working set (4), and removes extra pages when working set size is above the threshold (5).

the progress (i.e., amount of data sent/received over a time window) and estimate the bandwidth available to the application. Control is straightforward: if the application is seen to exceed its bandwidth threshold, it can be made compliant by just stretching out the data transmission or reception over a longer time period (e.g., by using fine-grained sleeps). The amount of delay is calculated so that the bandwidth at the end-point is not above the prescribed threshold. For clarity of description, we restrict our attention to synchronous communication operations and also assume that the data transmission rate in the network is not the bottleneck. The approach needs to be refined slightly to handle situations where communication operations are asynchronous.

Integrated and implicit resource usage

Applications do not access system resources in an isolated fashion. For instance, accessing a non-resident virtual memory page results in the triggering of an interrupt handler, transfer of a page from disk, accompanied with optional swapping out of a resident page and possible enlargement of the process working set size. To correctly handle such coupled accesses to system resources, we need to take into account effects such as increased CPU usage due to OS activity triggered on behalf of the application and additional disk usage because of reduced availability of physical memory pages.

Our sandboxing strategy factors in the above effects by appropriately defining the progress metric to reflect both *explicit* and *implicit* resource usage. The overall resource usage is forced to adhere to the requested limits by controlling the explicit requests. For example, even though an application's disk bandwidth usage due to paging is not controllable at the user level, its aggregate disk bandwidth usage can be reduced by controlling explicit disk requests such as file read and write. As a last resort, quantitative rate-based limits on resource usage can be enforced by controlling allocation of CPU resources to the application, therefore affecting its usage of all other resources.

5.1.2 Implementation on Windows NT

This section discusses NT-specific issues and demonstrates the control of CPU, memory, and network resources with experiments. The implementation and performance results below refer to NT 4.0, service pack 5, running on Pentium II 450 MHz machines.

Constraining use of CPU resources

Monitoring progress The CPU monitor is attached as a callback routine of the fine-grained *multimedia timers*, and is triggered every 10ms with high accuracy using a technique introduced in [29]. Note that the scheduling quantum on NT is at least 20ms for the workstation installation and 120ms for the server installation. The monitor obtains an application's CPU usage in terms of kernel time and user time through system API calls. The kernel time refers to the time the application is executing in kernel mode. However, this statistic does not account for all OS activity performed on behalf of the application. For instance, the overhead of memory paging is not included in per-process statistics, instead being recorded in the per-processor statistic. As a heuristic, we estimate the application's portion of this non-accounted kernel time by considering the ratio of the number of application events triggering such kernel activity (e.g., page faults) to the overall system-wide number of such events.

As described in Section 5.1.1, the monitor estimates process wait time within a time window by checking the process state and accumulating the time slots at which the process is found waiting. Although NT allows examining process state via its performance counter infrastructure, this incurs high overhead (on the order of milliseconds). Instead, we employ a heuristic that infers process state based on thread contexts. We observe that a thread can be in a wait state only when it is executing a function inside the kernel. Recognizing that if the thread is not blocked it is unlikely to stay at the same place in kernel code, the heuristic checks the instruction pointer register to see whether a trap instruction (int 2Eh) has just been executed, and whether any general registers have changed since the last check. If the same context is seen, it regards the thread as being in a wait state, with the process regarded as waiting if all of its threads are waiting.

Controlling progress Based on the progress metric, the controlling code decides whether or not to schedule the process in the next time slot. Although this decision could be implemented using OS support for suspending and resuming threads (which we use in our Linux implementation), the latter incurs high overhead. Consequently, we adopt a different strategy that relies on fine-grained adjustment of application process priorities to achieve the same result.

Our approach requires four priority classes (see Figure 5.3), two of which encode whether CPU resource are available or unavailable to the application. The monitor runs at the highest priority (level 4), and a special compute-bound "hog" process runs at a very low priority level 2 and executes only when no other normal applications are active. An application process not making sufficient progress is boosted to priority level 3, where it preempts the hog process and occupies the CPU. A process that has exceeded its share is lowered to priority level 1, allowing other processes (possibly running within their own sandboxes) or in their absence, the hog, to use the CPU. Note that this scheme allows multiple sandboxes to coexist on the same host.

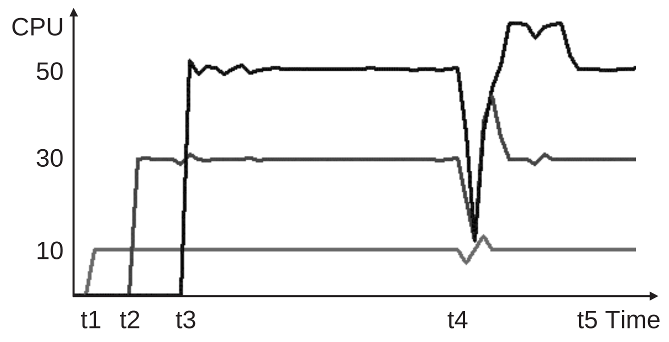
Effectiveness of the sandbox Our experiments show that this implementation enables stable control of CPU resources in the 1% to 97% range. When the requested share is above 97%, the measured allocation includes perturbations from background load (the performance monitor, system processes, and the sandboxing code). The interference from sandboxing code consists of the monitor overhead and bursty allocation of

Priority level	CPU available	CPU not available
4	`Monitor`	`Monitor`
3	**Application**	
2	`Hog`	`Hog`
1		**Application**

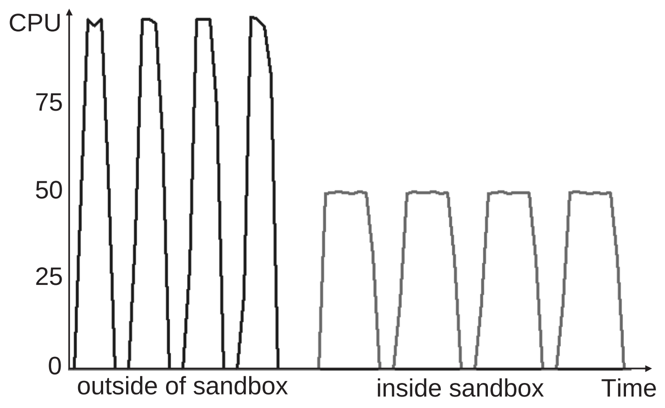
Figure 5.3: Controlling application CPU availability by changing process priorities.

resources to the hog process over long runs (this is an NT feature for avoiding starvation). The overhead of adjusting the priority is negligible. To measure the overall costs of running an application within a sandbox, we compare the wall-clock execution time of a synthetic CPU-intensive application running within and outside of a sandbox. On average, this application took 35.5 seconds to finish when running alone, and took 36.0 seconds when running inside a sandbox prescribing a CPU share of 100%, indicating an overhead of about 1.5%.

Figure 5.4(a) is a snapshot of the performance monitor display showing three sandboxed applications running on the same host. They start at times *t*1, *t*2, and *t*3, requesting 10%, 30%, and 50% of the CPU share, respectively. With the total CPU request at 90%, all three applications receive a steady CPU share until time *t*4, when we deliberately perturb the allocation by dragging a DoS window. This causes the total available CPU to decrease drastically (because of the kernel activity), and a sharp decrease in the immediate CPU shares available to each application. However, this drop is compensated with additional CPU resources once the system reacquires CPU resources (end of window movement). These results indicate that the sandbox can support accurate and stable CPU sharing with resilient compensation.



(a)



(b)

Figure 5.4: (a) Weighted CPU sharing for multiple applications. (b) Constraining CPU share for applications with wait states.

Figure 5.4(b) shows the execution of an application that sleeps periodically, without sandboxing (left) and with a sandboxed CPU share of 50% (right). The working time with the sandbox is twice the amount on the left, corresponding to the halved CPU resource. More importantly, the sleep (wait) time is kept the same, consistent with Figure 5.1 and verifying the effectiveness of our state-checking heuristic.

Constraining use of memory resources

Monitoring progress An API call, `GetProcessMemoryInfo`, provides information about the resident memory of a process. Unlike the CPU case, the sampling of this information can be adapted to the rate at which the application consumes memory resources. To estimate the latter, we integrate the sampling with the controlling scheme described below.

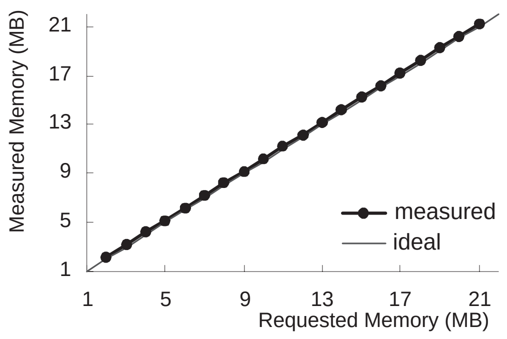
Controlling progress As described in Section 5.1.1, controlling progress of memory resources requires the sandboxing code to relinquish surplus memory pages to the OS. To do this, we rely on a convention in NT: pages whose protection attributes are marked `NoAccess` are collected by the swapper.

The same core OS mechanism, user-level protection fault handlers, is used to decide both (a) *when* a page must be relinquished, and (b) *which* page this must be. Our scheme intercepts the memory allocation APIs (e.g., `VirtualAlloc` and `HeapAlloc`) to build up its own representation of the process working set. When the allocated pages exceed the desired working set size, the extra pages are marked `NoAccess`. When such a page is accessed, a protection fault is triggered: the sandbox catches this fault and changes page protection to `ReadWrite`. Note that this might enlarge the working set of the process, in which case a FIFO policy is used to evict a page from the (sandbox-maintained view of the) working set. The protection fault handler also provides a natural place for sampling the actual working set size, since a process's consumption of memory is reflected by the number of faults it incurs.

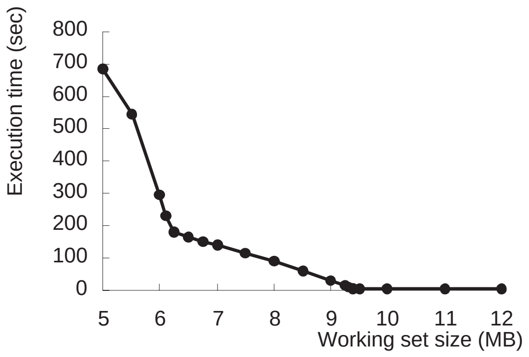
A few additional points need clarification. The implementation is simplified by not evicting pages containing executable code, so this limits the least amount of memory that can be constrained. Eviction at the sandbox level may or may not cause the page to be written to disk although these pages are excluded from the process working set; when the system has large amounts of free memory, NT maintains some pages in a transition state delaying writing them to disk. Note that with our design, if the application is running within its memory limits, it will not suffer from any runtime overhead (except that of intercepting API calls). Beyond that point, the overhead are a function of process virtual memory locality behavior.

Effectiveness of the sandbox Our experiments show that, on a 450 MHz Pentium II machine with 128MB memory, this sandbox implementation can effectively control actual physical memory usage from 1.5MB up to around 100MB. The lower bound marks the minimal memory consumption when the application is loaded, including that by system DLLs. To compare, a “Hello, world” program consumes about 500KB memory and one that creates a TCP socket consumes 1MB memory. The upper bound approximates the maximum amount of memory an application can normally use in our system. The memory overhead includes 64KB for the code injected into application address space and 12 bytes for keeping track of each page in the working set. The overhead of intercepting a memory allocation call is measured as 1.07μs when the specified memory constraints are above the working set size (thus no page fault is incurred). When the constraints are below the required working set size, process memory locality behavior determines the overhead. However, because of our CPU accounting scheme, only this process’s execution time is affected.

Figure 5.5(a) shows the requested and measured physical memory allocations for an application that has an initial working set size of 1.5MB and allocates an additional 20MB of memory. The sandbox is configured to limit available memory to various sizes



(a)



(b)

Figure 5.5: (a) Controlling the amount of physical memory utilized by an application. (b) Execution time as size of working set varies.

ranging from 2MB to 21MB. As the figure shows, the measured memory allocation of the application (read from the NT Performance Monitor) is virtually identical to what was requested.

Figure 5.5(b) demonstrates the impact of the memory sandbox on application execution time. The application under study has a memory access pattern that produces page faults linearly proportional to the non-resident portion of its data set. In this case, the application starts off with a working set size of 1.5MB and allocates an additional 8MB. The sandbox enforces physical memory constraints between 5MB and 12MB. As the figure shows, the execution time behavior of the application can be divided into three regions with different slopes. When the memory constraint is more than 9.5MB, all of the accessed data can be loaded into physical memory and there are no page faults. When the memory constraint is below 9.5MB, total execution time increases linearly as the non-resident size increases, until the constraints reaches 6.25MB. In this region, page faults occur as expected but the process pages are not written to disk. When available memory is below 6.25MB, we observe heavy disk activity. In this segment, the execution time also varies approximately linearly, with the slope determined by disk access characteristics. These experiments show that our sandboxing scheme preserves the application page faulting behavior.

Constraining available network bandwidth

Monitoring and controlling progress As described in Section 5.1.1, we intercept socket APIs (*accept*, *connect*, *send*, and *recv*) to monitor and control available network bandwidth.

Effectiveness of the sandbox The effectiveness of the sandbox is evaluated on a pair of Pentium II (450 MHz) machines connected to a 10/100 auto-sensing Fast Ethernet hub. The application consists of a server and one or more clients in a simple ping-pong communication pattern exchanging 4KB-sized messages, which achieves a peak bandwidth of 11 MBps when running outside the sandbox. Our experiments show that the sandbox can effectively constrain bandwidth from 1 Bps to about 11 MBps with an error of less than 2%. Figure 5.6 shows how accurately the sandbox controls network resources for constraints in the range 0.5 KBps to 8 MBps.

5.1.3 Differences in Linux implementation

Linux provides support very similar to Windows NT for instrumenting application binaries, and monitoring and controlling resource consumption. For instance, library functions such as the sockets and memory allocation APIs, can be intercepted by preloading shared libraries. The mechanisms and performance of network bandwidth control are identical across the Windows NT and Linux platforms. However, there are small differences in how CPU and memory resources are constrained under Linux.

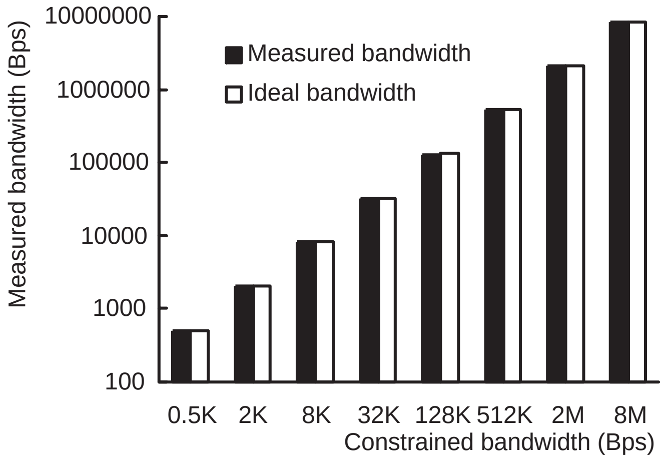


Figure 5.6: Comparison of application observed network bandwidth and the ideal values under the control of sandbox.

Controlling CPU resource Adjusting scheduling priorities requires superuser privilege on Linux. Therefore, we use a scheme based on thread suspend/resume: the sandbox sends an application a SIGSTOP signal to suspend and a SIGCONT signal to resume its execution.

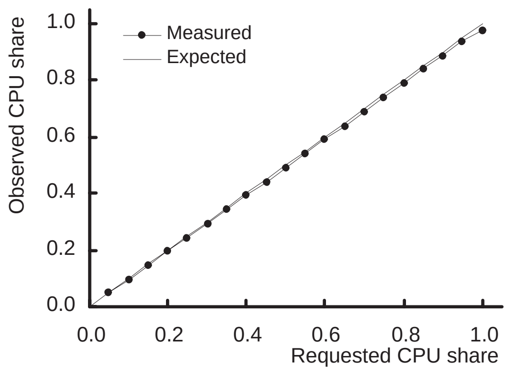
Controlling memory resource Linux provides a `setrlimit` API for limiting the maximum amount of physical memory a process can use. However, current versions of the kernel (e.g., v2.2.12) do not enforce this constraint. Consequently, we adopt a scheme similar to the one on NT. However, unlike on NT, where an implicit protocol (using `NoAccess` protection bits) between the OS and the application permits the former to collect pages not required by the latter, no such protocol exists on Linux. The page protection bits can be set on Linux, but the kernel swapper (`kswapd`) does not check the page attributes to decide which page must be swapped out.

We get around this problem by handling the swapping ourselves. First, we intercept memory allocation functions (e.g., `malloc`) to make sure that only the requested amount of physical memory is kept valid; all other memory pages are protected to be unavailable for access. When a page fault happens due to invalid access, we pick another page (in FIFO order) from the resident set (maintained by the sandboxing code), save its contents to our own swap file, and take it out of the resident set using the `munmap` mechanism in Linux. Subsequently, an invalid access exception triggers the saved contents to be mapped back to the original virtual address.

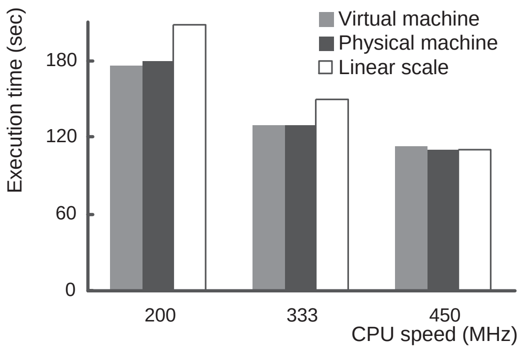
5.2 Obtaining Application Performance Profiles

Since our design of the sandboxing tool enables the control of application resource usage as if only the prescribed amount is available to their execution, we can use it to emulate different resource scenarios. This usage requires that the resource levels be accurately controlled and that application behavior on this virtual execution environment approximate executions on real distributed environments.

Figure 5.7 demonstrates the overheads and accuracy of the virtual execution environment. Figure 5.7(a) compares the execution time of a tight loop on the virtual execution environment (realized on a Pentium II 450MHz machine) and the expected execution time (normalized with the requested share) as CPU share varies from 5% to 100%. The application's execution time under the virtual execution environment is very close to what is expected (except when 100% CPU share is requested) indicating that the environment incurs low overhead. Figure 5.7(b) shows the accuracy of the virtual execution environment by comparing execution times for the active visualization application on different real client machines (a Pentium Pro 200MHz, a Pentium II 333MHz,



(a)



(b)

Figure 5.7: (a) Overheads and accuracy of the virtual execution environment with a tight loop. (b) Comparison of application execution times using the virtual execution environment and physical machines for the active visualization application.

and a Pentium II 450MHz) with that obtained under the virtual environment when the latter is configured to provide a CPU share based on the WinBench 99 v1.1 [22] scores for the different machines. WinBench is a Windows 98/NT benchmark that measures the aggregate impact of CPU speed and memory configuration. The execution times under the virtual environment are within 3% of that on the physical machines. Note that similar accuracy cannot be obtained by simply scaling the execution time in proportion to the CPU shares based on the WinBench benchmark. This estimate, shown by the “Linear scale” bar, turns out to have a big difference from real executions (primarily because of idle time behavior), indicating the necessity to measure application behavior under different resource conditions.

Given such sandbox environments that accurately emulate differing resource availability, obtaining profile-based application behavior is straightforward. A driver program executes each configuration repeatedly setting up the virtual execution environment to sample different resource conditions. A separate tool analyzes output quality measures to determine configurations and regions of the resource space that require additional samples. The output of this modeling step is a *performance database* that records information about a maximal subset of the configurations representing the resource profile of this application, which can informally be defined as configurations that outperform other configurations under at least one resource situation. Additionally, configurations that exhibit similar execution behavior can be merged (with only one of them being stored) in the performance database. These measurements are interpolated to get performance curves that summarize configuration performance.

The performance database for an application contains its resource profiles, indicating the application performance (in terms of QoS metrics) for various application configurations (in terms of the setting of control parameters) under various resource conditions. Ideally, this mapping should be expressed in an analytic model. However, it may not be easy to fit the data to exact expressions. The performance database currently takes a simplified view in storing all the discrete profiling records and performing simple linear interpolation to get estimates of application performance under unspecified resource conditions.

Currently, the performance database is stored as a list of records, as shown in Figure 5.8. Each record contains a list of name value pairs, with each name being either a control parameter, a resource property, or a QoS metric.

As an extension, the performance database could also include the measured variance, analytical expressions that approximate the database, or other analysis results such as the calculation of bottleneck resources for a specific scenario.

The profiling process for obtaining the performance database could be exhaustive

```
control.fovea=80   resc.cpu=0.4  qos.response_time=780  ...
control.fovea=80   resc.cpu=0.9  qos.response_time=400  ...
control.fovea=320  resc.cpu=0.4  qos.response_time=1500 ...
control.fovea=320  resc.cpu=0.9  qos.response_time=750  ...
```

Figure 5.8: A simple form of performance database.

or selective. The former experiments with various kinds of resource conditions and their combinations in a uniform fashion. The latter is more selective and can analyze the resource ranges that needs more experiments and avoid profiling that do not generate useful information. We currently do not have the sensitivity analysis tool and use exhaustive profiling to get the performance database for applications in Chapter 8 and Chapter 9.

5.3 Summary

To model performance of various configurations of tunable applications, we have designed a virtual execution environment that controls resource availability to applications and implemented it on Windows NT and Linux. We use driver programs to enumerate application configurations and control resource conditions to profile application behaviors under various resource conditions. The results are stored in a performance database which serves information for selecting appropriate configurations at run time.

Chapter 6

Run-time Adaptation System

At run time, an appropriate application configuration is chosen and dynamically updated as resource availability changes to satisfy user preference constraints (i.e., *QoS constraints*). Such configuration and adaptation are realized by interactions between three components as in Figure 6.1: (1) an *application-specific monitoring agent* that monitors resource characteristics of interest to the application as well as application progress, (2) a *resource scheduler* that correlates observed resource characteristics and user preferences with performance models stored in the performance database, and (3) a *steering agent* that provides a control message interface and performs the actual reconfiguration.

The steering agent inside the application understands the name and data type of the tunability parameters and acts as a bridge between the monitoring agents (including external ones), the resource scheduler and the application. It asks the resource scheduler to find a matching configuration when obtains a changed resource condition, in terms of the setting of control parameters for the selected configuration and expected performance level (i.e., QoS metrics). It sets the control parameters and resource con-

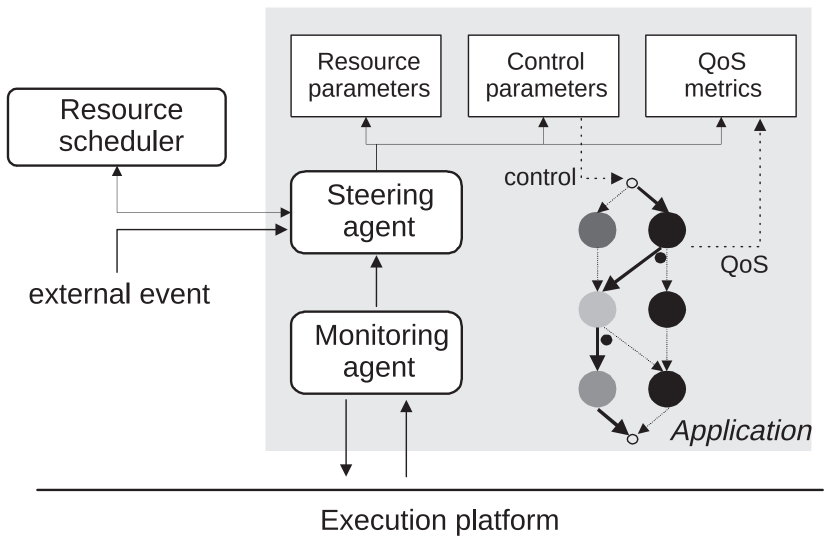


Figure 6.1: Run-time components and tunability data structures.

ditions accordingly, and records the expected values of QoS metrics. When application entering a tunable component or encountering a tuning point in a component, control parameters are mapped to the local variables they correspond to, thereby controlling the behavior of execution. At the end of a component execution, the values of QoS metrics are recorded by the generated inline code, which could be potentially combined with expected QoS levels to get a dynamic feedback about application execution (currently we are not performing the combination).

6.1 Monitoring Agent

The *monitoring agent*, running periodically as an application module, continually observes application progresses and estimates the fraction of resources that are available for use by the application. The monitoring agent relies on a system database for information about maximum capacities of system resources (e.g., CPU speed, physical memory pages, and network bandwidth). It observes the shortfall between the level of resources requested by the application and what it actually obtained: comparing allotted CPU time with the wall clock time (factoring in periods where the application is waiting), comparing physical memory usage with virtual memory size, and keeping track of aggregate network traffic over time. The results are estimates about resource availability.

The monitoring agent uses the same techniques as in the virtual execution environment described in Chapter 5. Our experience shows that this data collection adds negligible overhead to application execution even when performed in very fine-grained time slots (e.g., 100ms). The monitoring agent retrieves the CPU usage information from the operating system, using debugging APIs (e.g., `GetProcessTimes`) on NT (or `/proc` filesystem or Linux). It takes into account the history of obtained CPU usage information of certain period and the CPU speed to compute an estimated CPU availability. Figure 6.2 shows the CPU shares the monitoring agent observes as that is changed by the virtual execution environment from 90% to 40% at time 30 seconds. Although the CPU share is initially assumed to be 100% by the monitoring agent, it correctly estimates a 90% level after the first several seconds. This period could be reduced with the help of an external agent informing resource availability at startup time. When the CPU share drops to 40% at time 30 seconds, the monitoring agent realizes the change immediately, even though it takes several seconds to obtain the 40% level estimate. The length of this period depends on the size of the history window from which an average is computed. The larger the windows size, the more stable the estimate is; however, the slower the monitoring agent realizes the changes in resource availability.

For memory resource, the monitoring agent uses the OS mechanisms to obtain the

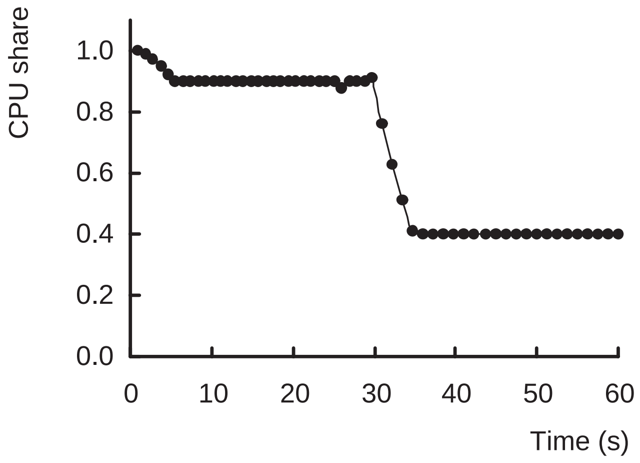


Figure 6.2: Monitoring agent's observed CPU share as the actual share is changed from 90% to 40% at time 30 seconds.

usage of physical memory (i.e., the size of working set): on NT via performance monitoring API `GetProcessMemoryInfo`, on Linux via the `/proc` filesystem. It intercepts memory allocation/deallocation calls (e.g., `malloc`) to keep track of the size of virtual memory the application requests. The amount of physical memory available to this application is estimated to be the size of the working set when the virtual memory size is above the physical memory size and the latter ceases to increase for a while.

For network resources, the monitoring agent intercepts networking API calls such as `send` and `recv`, and measures the amount of data transmission as well as the time spent on these operations. It accumulates this information to estimate the available network bandwidth. Figure 6.3 shows the observed network bandwidth as the actual bandwidth is changed by the virtual execution environment from 500 KBps to 50 KBps at time 30 seconds, assuming the monitoring agent is informed of the exact resource level in the beginning. The monitoring agent is able to estimate correctly at both of the

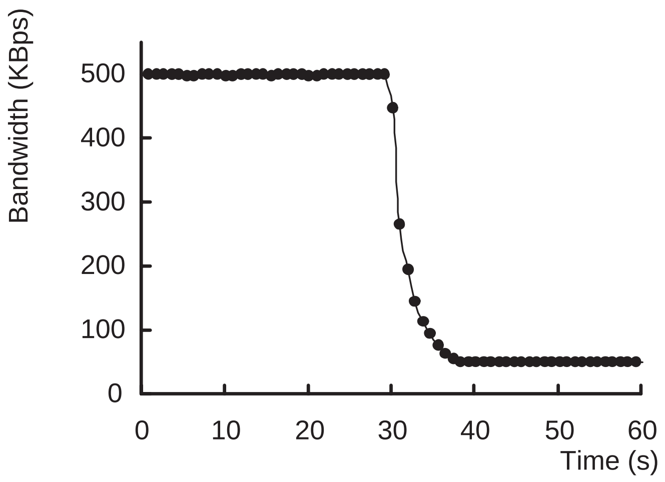


Figure 6.3: Monitoring agent's observed network bandwidth as the actual bandwidth is changed from 500 KBps to 50 KBps at time 30 seconds.

two resource levels. Again, its responsiveness to resource changes depends on the size of the history window maintained for calculating an average.

It is possible that monitoring of these resources is done outside the applications. We put it inside application to get a better estimate of the resource levels available to a specific application (since it can run on a virtual execution environment), instead of resource availability to the entire host.

Resource availability is measured in a specialized thread by the monitoring agent, asynchronous to an application's execution. In contrast, the measurement of application progress by the *QoS monitor code* (in terms of QoS metrics) is inline, synchronous to application execution and is usually application-dependent. The latter can also be thought of as part of the monitoring process since it can also result in events to which the application can adapt. In addition, the application is open to receiving external notifications through which an outside monitoring entity could inform the change of additional conditions. This allows plugin of arbitrary monitoring tools through the use of the control message interface.

6.2 Steering Agent

The *steering agent* is responsible for switching application configurations, executing any clean-up code as appropriate. In addition, the steering agent sends and receives control messages, which specify new values for control parameters, the resource conditions under which these new settings are valid, and the expected performance level (in terms of QoS metrics). Upon receiving them, the steering agent sets up the new values for control parameters, which would be suggested to the application at the entrance of a component or a transition function, as specified by the tunability interface (see Chapter 4).

In implementation, the steering agent runs as a separate thread in application. The control messages are realized by reliable UDP messages. The steering agent understands these messages because the parameters are described in the tunability interface. After these control messages are received, it sets the control parameters to the new values and resets the conditions that trigger performance exception.

6.3 Resource Scheduler

The resource scheduler takes as input the QoS constraints, the performance database, and the resource conditions, and computes an appropriate application configuration to select for execution. Furthermore, different scheduling policies could be implemented to optimize user satisfaction, application performance, or system resource utilization.

Schedulers can be divided into two categories: schedulers that only concern the performance of a single application and schedulers that aim at optimizing a global metric. Both of these schedulers could be plugged into our framework provided they follow the same interface: receiving of control messages about the execution conditions and user preferences and sending of control messages describing a new configuration. The control messages are a list of setting of tunability parameters (i.e., control parameters, resource parameters, and QoS metrics) where the setting of each parameter contains its name, a value, and possibly an value range within which the new configuration is considered stable (such as for ranges of resource parameters).

As an input to resource scheduler, each user preference constraint is expressed as the value ranges on a subset of output quality metrics and is accompanied with an objective function to be optimized. For simplicity, we assume a relatively restricted form of this function: maximizing or minimizing a single quality metric. These constraints when considered together with measured resource characteristics, restrict the suitable set of application configurations. Of these, the resource scheduler picks the one that best satisfies the objective function. Multiple user preference constraints can be specified. The system examines them in decreasing order of preference; in the case that one request cannot be satisfied due to inadequate resources, the system attempts fulfill the next preferred constraint. When dispatching multiple application executions with the corresponding user preference requests, the scheduler could prioritize preferences from different users, implementing its own policy in selecting the performance levels for each request.

QoS constraints in Figure 6.4 shows a user preference constraint, specified using QoS metrics `response_time`, `transmission_time`, and `image_level` that are defined in the tunability interface. It requires that `response_time` to be in the range of 0 to 1000, `execution_time` to be in the range of 0 to 100000, and `image_level` to be in the range of 3 to 5. It also asks to maximize the `transmission_time` metric.

```
[low]  qos.response_time = 1000
[high] qos.response_time = 0
[low]  qos.transmission_time = 100000
[high] qos.transmission_time = 0
[low]  qos.image_level = 3
[high] qos.image_level = 5
[maxi] qos.transmission_time
```

Figure 6.4: Specifying user preferences in QoS constraints.

A further extension of QoS constraint format could allow users to specify their own adaptation policy. For instance, introduction of symbolic expressions of QoS metrics permits users to specify more complicated objective functions or constraints. Although constraining application performance metrics might be a reasonable request for some end-users, it might not be practical for ordinary users. For these users, a potential extension can provide several classes of default settings of QoS constraints and assign the users to one of these classes.

The schedulers for the Active Visualization and Junction Detection applications used in Chapter 8 and Chapter 9 are manually written, with the first concerning the performance of a single application and the second aiming at optimizing a global metric. The latter handles multiple applications of the same kind (i.e., same parameters in the tunability interface).

A generic scheduler for tunable applications should satisfy the following form: It takes as inputs three kinds of information for each application (1) a template generated for this application specifying its parameters in the tunability interface, including control parameters, environment parameters, and QoS metrics; (2) a performance database generated in the profiling stage describing application performance, in terms of the mapping from control parameters, environment parameters, to QoS metrics; (3) a user preference for executing the application in the form of QoS constraints. The template information helps resource scheduler interpret the latter two kinds of information. For instance, it tells whether the increment of a specific QoS metric means improvement or degradation of application performance. In addition, a generic scheduler must export an interface similar to steering agents, responsible for receiving request from steering agents and replying with a certain configuration (i.e., a row in the performance database) it chooses based on some scheduling policy. This interface allows schedulers to cope with adaptation decision of different applications because the semantics of tunability parameters are hidden from the schedulers.

6.4 Summary

In our framework, application adaptation is performed by run-time components: a monitoring agent, a resource scheduler, and a steering agent, interacting with each other through control messages that exchange information about parameters exposed in the tunability interface. The monitoring agent is responsible for obtaining resource conditions, and the resource scheduler for making adaptation decisions. The steering agent acts between the application, resource scheduler, and monitoring agent, responsible for realization the adaptation decisions. This infrastructure can be applied to many applications that is originally oblivious of adaptation to achieve a better performance or user satisfaction.

Chapter 7

Benefits of Tunable Applications

Application tunability provides flexibility in application execution that allows a resource scheduler to find the configuration best matching the current system conditions. As a result, we expect both the performance of the application and the resource utilization of the system to be improved. To understand the performance impact of tunability, we study a particular system and characterize the benefits of tunability using simulation. Although tunability allows complicated tradeoffs among many types of resources in a dynamically changing environment, for simplicity, we limit the resource type to be only processors in a relatively static scenario with a fixed number of processors, reflecting processor allocation in Calypso system [3]. The only dynamic factor is the arrival of applications requesting for processor resources. We compare the system utilization achieved with tunable parallel applications against that with non-tunable applications. We first propose a simple greedy heuristic for scheduling tunable parallel applications with predictability requirements, and then use a parameterizable task system to systematically quantify the benefits and shortcomings of tunability.

7.1 Scheduling Formulation

Without tunability, the underlying scheduling problem we address is one of dynamically scheduling parallel real-time jobs (i.e, applications) in a system with a fixed amount of homogeneous processing resources. We restrict our attention to jobs which can be represented as a chain of tasks where each task has an associated deadline. A task is a schedulable unit, corresponding to a component in parallel applications. Each task is assumed to be non-preemptible and should not be scheduled at all if the deadline would be missed. A job should not be scheduled if one of its tasks misses the deadline. Here, tunability represents the flexibility in the number of processors that can be allocated to each task as well as the maximum number of processors the task can utilize (i.e., degree of concurrency). We study tunability benefits for both non-malleable (in Section 7.3) and malleable (in Section 7.4) executions of these tasks. Non-malleable tasks, characteristic of most current-day parallel applications written in a message-passing style using systems such as PVM [26] or MPI [30], require a fixed resource "shape" in terms of the number of processors required over a time period. In contrast, malleable tasks such as in Calypso [3] programs, can execute on any number of processors up to their degree of concurrency. In both cases, we assume that information about all tasks of the job is available upon job arrival. The primary objective of this scheduling problem is to maximize the number of on-time jobs. A secondary objective is to maximize system utilization.

With tunability, the only change to the scheduling problem is that a job is now represented by an OR task graph instead of a chain. The multiple paths in the task graph represent various alternate executions of a tunable program. For uniformity, we assume that all paths through an OR graph have been enumerated, so a tunable application is represented by multiple task chains. For the purpose of this study, we assume that each chain requires the same total amount of resources and achieves the same output quality. Note that in practice, task chains of a tunable application are likely to have different overall resource requirements and output qualities: the issue then is to maximize the achieved output quality.

As with most non-trivial scheduling problems, all of the above formulations are NP-hard. Even if the original problem is not, adding tunability would probably turn it into. For instance, feasibility test of deadline-constrained sequential jobs on a uniprocessor is polynomial [20]; however, the same test for tunable jobs can be reduced from Knapsack problem and therefore is NP-hard. We next describe a simple greedy heuristic for the above NP-hard scheduling problem.

A Simple Greedy Heuristic The heuristic allocates resources to jobs using a first fit policy. For a tunable job with multiple schedulable configurations, the heuristic finds among all of them the one that so far most efficiently uses the system. The heuristic keeps track of available *maximal holes* in the processor-time 2D space: each hole is represented by a triple (m, t_b, t_e) (denoting that m processors are available from beginning time t_b until the end time t_e), and is maximal if it is not contained within any other hole. A job is *schedulable* if all the tasks on its task chain (any one of the task chains for a tunable job) can be scheduled into available holes while meeting the task deadlines. Ties between schedulable configurations are broken in favor of chains which maximize system utilization (over a time window defined by the job's release time and scheduled finish time) and require fewer total resources for some prefix of their tasks. Under the assumptions of our task model, the heuristic finds the job configuration which achieves the earliest finish time.

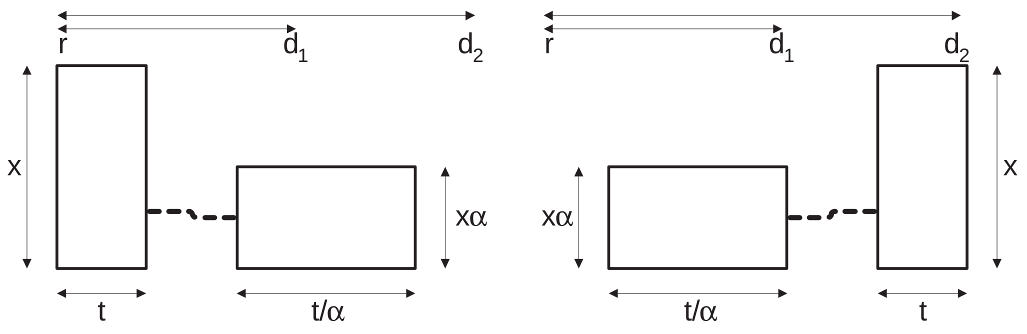


Figure 7.1: A parameterizable tunable job. The job parameters x, α, and laxity, permit the convenient simulation of a range of job shapes and deadline characteristics.

7.2 A Parameterizable Task System

To systematically explore the space of application behaviors, we consider a task system that consists of a parameterizable tunable job shown in Figure 7.1. The job parameters enable the convenient simulation of a range of job shapes and deadline characteristics.

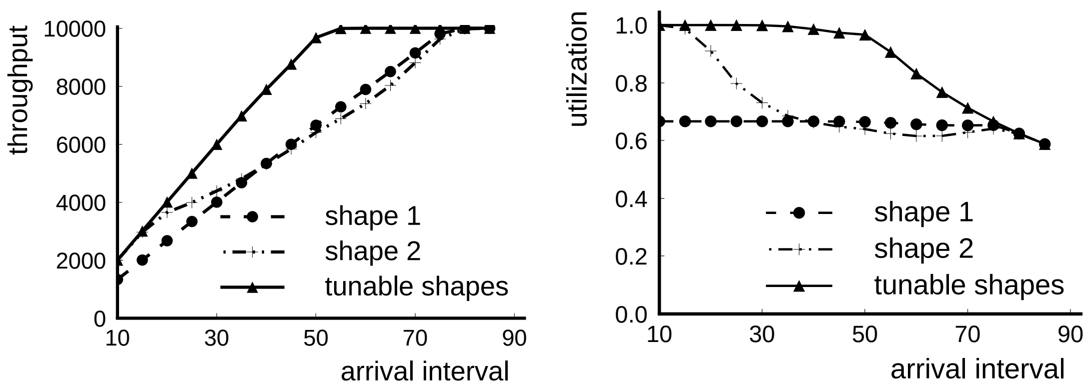
The job consists of two chains, each with two tasks. The two configurations simply transpose the positions of the two tasks. Each task requires the same total amount of resources but with different shapes. One task asks for x processors for time t, whereas the other task requests $x\alpha$ processors for time t/α amount of time. The value of α is chosen in the interval $(0, 1]$ such that both x and $x\alpha$ are integers. Modifying x and α allows the simulation of a variety of task shapes. The task deadlines are set in terms of another parameter, the *laxity* of the job, expressed as the ratio of the slack time in the time period from the release time to the deadline. For a job released at time r, the deadline of the first task is set to $d_1 = r + \max(t, t/\alpha)/(1 - \mathrm{laxity})$; the deadline of the second task is set to $d_2 = r + (t + t/\alpha)/(1 - \mathrm{laxity})$. Since a task can begin execution as soon as its immediate predecessor completes, the task deadline denotes the time by which the task and all its predecessors must finish. The laxity parameter allows the systematic modeling of different amounts of slack time, and hence the constraints that exist for "fitting" a job into the system.

With the above parameterizable job, we construct three task systems: the first task system is tunable, consisting of both job configurations, while the other two systems are non-tunable, containing one configuration apiece. Left profile of Figure 7.1 is referred to as shape 1 whereas the right one referred to as shape 2. Jobs in each task system arrive according to the Poison distribution. We quantify the benefits of tunability in terms of two metrics—*system utilization* and *job throughput*—measuring the performance of the tunable task system as compared to the non-tunable task systems as a function of four parameters: *mean arrival interval*, *laxity*, the *number of processors* in the system, and α which controls the job shape. All experiments reported in this section assume x=16 processors, t=25 time units, and 10,000 job arrivals.

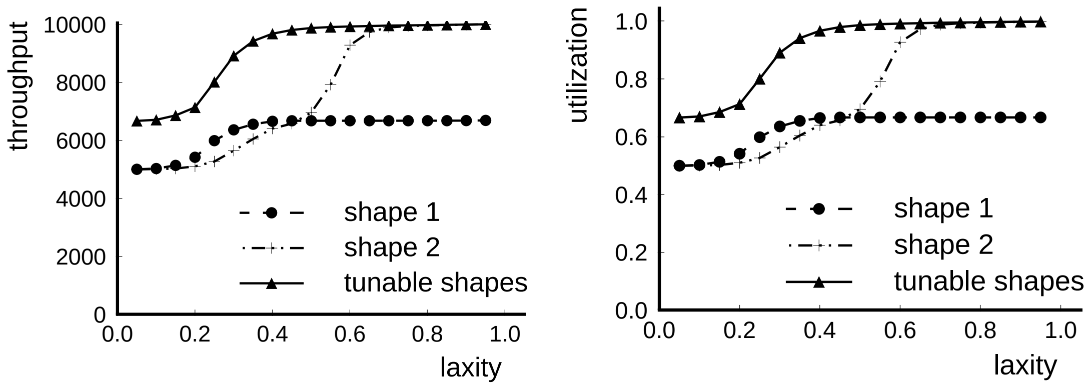
7.3 Tunability Benefits for Non-malleable Tasks

We first study the benefit of tunability for jobs consisting of tasks that require a fixed number of processors over a period of time. Figure 7.2 shows the system utilization (left) and throughput (right) as the mean arrival interval, the laxity, the number of processors, and the job shape are varied one parameter at a time, keeping the others fixed.

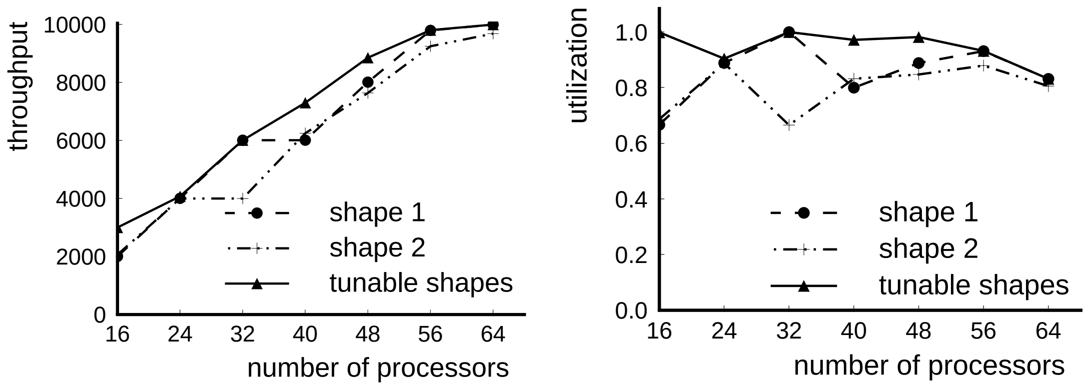
Sensitivity to inter-arrival time In Figure 7.2 (a), the arrival interval varies from 10 to 85 units (note that t=25), with the other parameters fixed as follows: *number of processors*=16, *laxity*=0.4, and α=0.5. When the arrival interval is small, the system is overloaded and only a small portion of the tasks are admitted in all three systems.



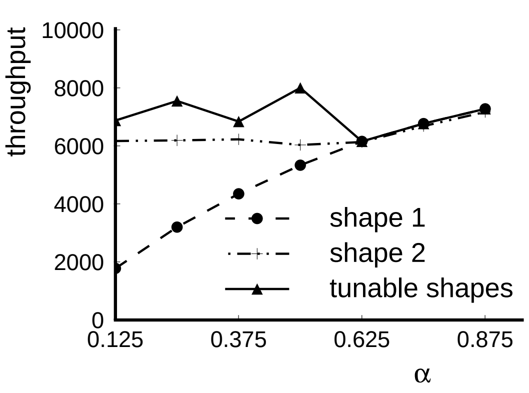
(a) Variation in average arrival interval

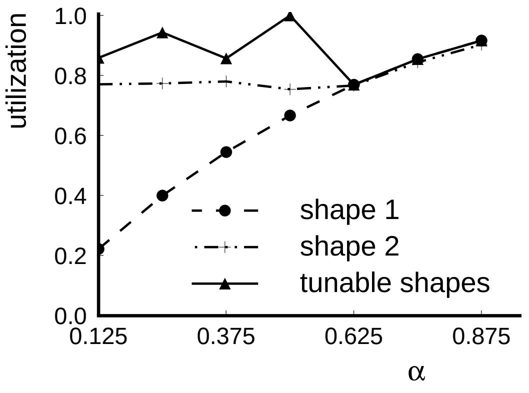


(b) Variation in laxity



(c) Variation in number of processors





(d) Variation in shape

Figure 7.2: Performance impact of tunability for non-malleable tasks as average arrival interval, laxity, number of processors in the system, and job shapes are varied, in terms of throughput (left) and system utilization (right).

Tunability has negligible performance impact, since the system is already being fully utilized. When the arrival interval is very high, the system is underloaded and all three task systems can admit all the jobs. Tunability does not yield much benefit since resources are abundant compared to requests. It is in the middle range of arrival intervals however, that the tunable system achieves the largest improvement in both utilization and throughput: at its peak, it can admit 3000 more jobs and achieve 30% better system utilization. The tunable system can decide which of the task configurations to use based on resource availability, resulting an efficient utilization of resources.

Sensitivity to laxity In Figure 7.2 (b), the laxity varies from 0.05 (a slack time of 5%) to 0.95 (a slack time of 20 times the processing time), with the other parameters fixed as follows: *number of processors*=16, α=0.5, *mean arrival interval*=50 time units. If there are no timeliness requirements (i.e., laxity is 1), all jobs would be admitted and the resources are fully utilized. In the presence of timeliness requirements, some jobs would be rejected because their deadlines cannot be met. When laxity is small, the job deadlines are tight and because there is little processor-time space left to fulfill resource requirements, the tunable system yields only a small improvement. However, this improvement goes up with an increase in laxity, decreasing to zero only when there is enough space to admit even non-tunable tasks. For shape 2, this happens when the laxity is above 60%. In contrast, shape 1 requires a larger number of processors for its first task, preventing it from a good packing (due to the greedy nature of the heuristic) even when deadlines are loose. The latter situation demonstrates the performance handicap of an inflexible (i.e., non-tunable) application.

Sensitivity to the number of processors Figure 7.2 (c) shows the benefits of tunability when the number of processors in the system are increased from 16 to 64 (recall that x=16), with the other parameters fixed as follows: α=0.5, *mean arrival interval*=15 time units, and *laxity*=0.25. Throughput increases as more processors become available. However, for some values, the non-tunable systems fail to gain from more processors, resulting from a drop in resource utilization. In these cases, the packing of job shapes does not fit the resource capacity well and the extra resources are simply wasted. The tunable system diminishes the penalty of this effect, yielding robust performance. When the number of processors grows significantly larger than individual task concurrency, even non-tunable jobs are able to utilize the available resources well; consequently, the benefits from tunability decrease.

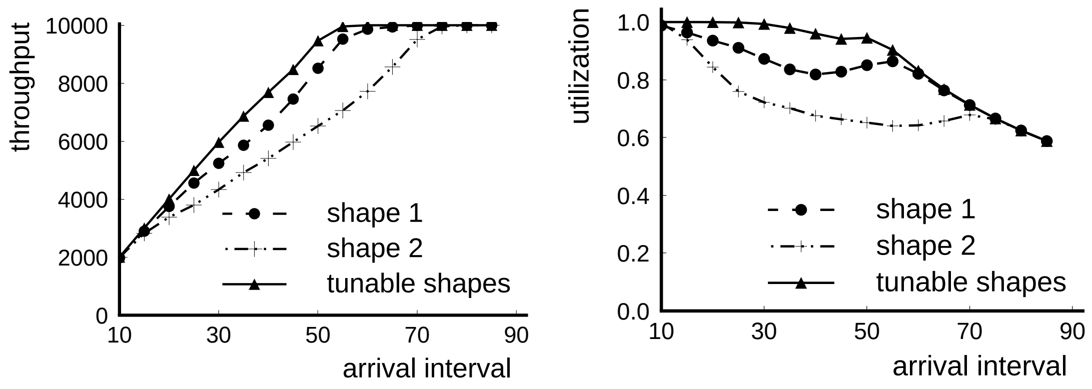
Sensitivity to the job shape Figure 7.2 (d) shows the benefits of tunability as a function of the job shape, determined by α. The other parameters are kept fixed as follows:

number of processors=16, *mean arrival interval*=40, and *laxity*=0.5. The utilization achieved by a particular job shape depends upon how well the shape can be packed given the number of processors in the system. Consequently, even when two job shapes have the same total resource requirement, they can yield very different overall system utilization because one of the shapes might be a better fit. We see that shape 1 performs far worse than the other two when α is small. Due to the suboptimal nature of the heuristic, a lot of resources are wasted for shape 1 jobs, proportional to the value of α. When α is not too large (up to 0.625), tunability improves performance achieving better packing than either of the individual shapes. For values of α that produce similar task resource profiles, as expected, tunability yields few benefit.

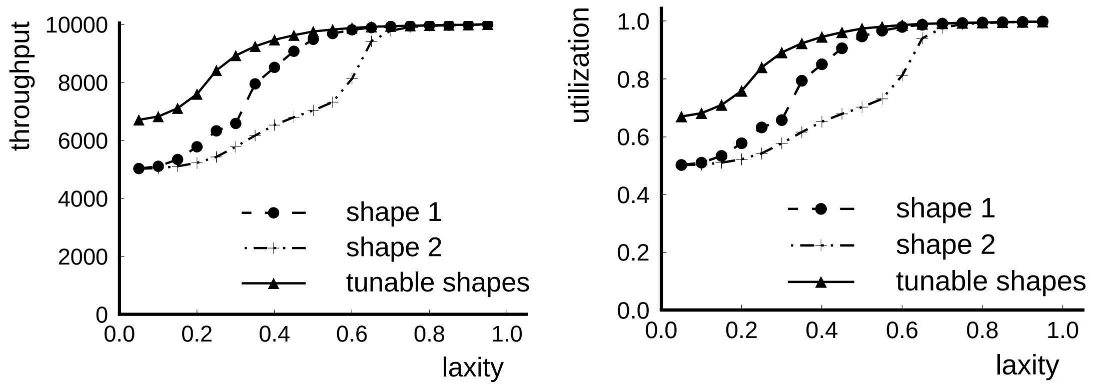
7.4 Tunability Benefits for Malleable Tasks

We conducted the same set of experiments for tasks that can be executed in a malleable fashion. These experiments, whose results are shown in Figure 7.3, differ from the non-malleable execution model only in that, instead of requiring a fixed task shape, they permit the task to change its shape to any area-preserving rectangle with a height less than its degree of concurrency. When allocating resources to a malleable task, the heuristic is extended to try various configurations of the task, starting from the highest number of processors the task can use (which corresponds to the shortest execution time).

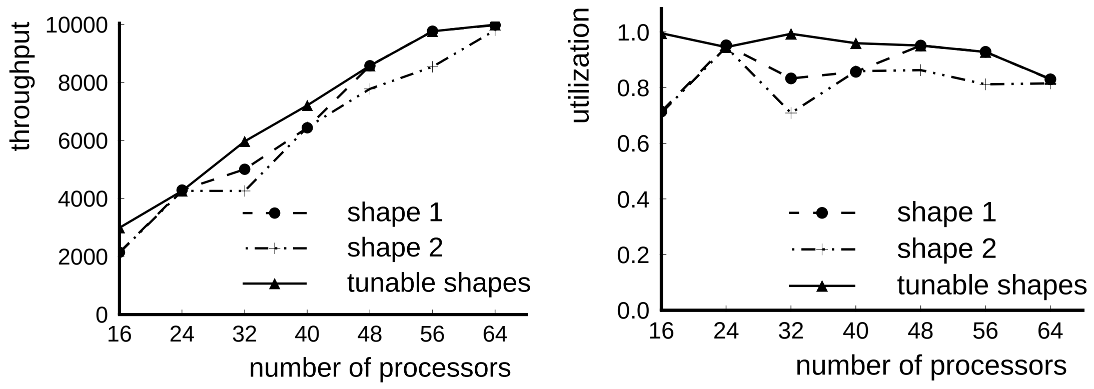
Comparing the results in Figure 7.3 and Figure 7.2, we find that both the tunable system and the non-tunable shape 2 job system perform similarly for both non-malleable and malleable models. What is noticeable however, is that the performance of the non-tunable shape 1 job system improves by a significant amount. Malleability ensures



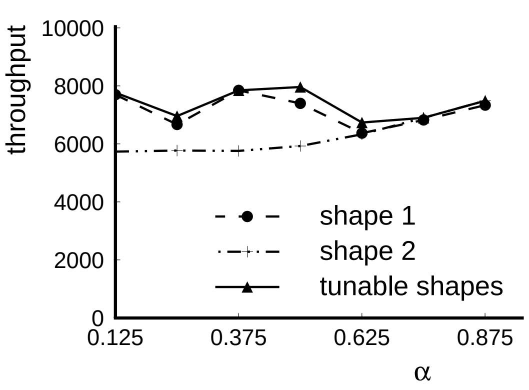
(a) Variation in average arrival interval

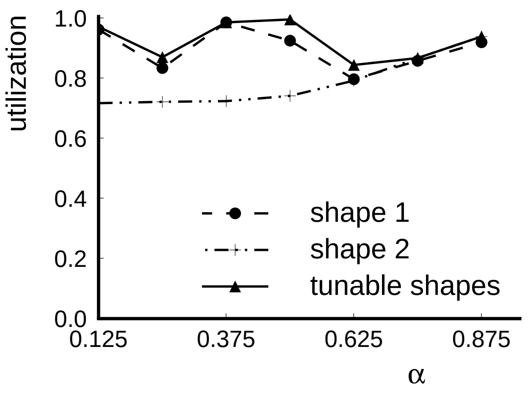


(b) Variation in laxity



(c) Variation in number of processors





(d) Variation in shape

Figure 7.3: Performance impact of tunability for malleable tasks as mean arrival interval, laxity, number of processors in the system, and job shapes are varied, in terms of throughput (left) and system utilization (right).

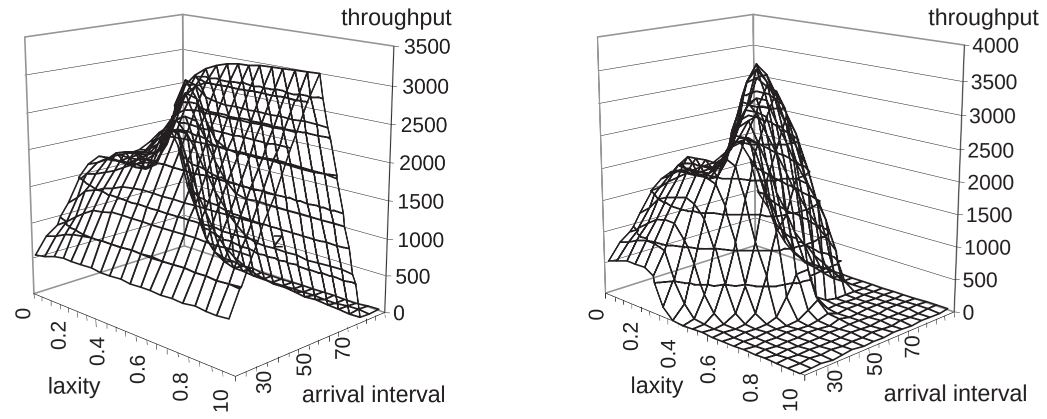
that these jobs achieve better packing as compared to their non-malleable versions, improving system utilization. Based upon the results shown in Figure 7.3, the same conclusion can be drawn regarding the overall benefits of tunability for malleable tasks as for non-malleable tasks, although the magnitude of the benefit might be smaller. In some situations, the sub-optimality of the heuristic prevents the tunable system from performing as well as one of the non-tunable systems. For example, in Figure 7.3 (d) when α=0.375, shape 1 packs slightly better than the tunable system, although the difference is negligible.

The reduced magnitude of improvement from tunability for malleable tasks might suggest that support for malleable tasks alone might be sufficient to improve overall system utilization. While this is certainly true for a range of parameter values, tunability complements malleability over a large range. Figures 7.4 (a) and 7.4 (b) show the increased throughput attained by the tunable job system when compared with the non-tunable shape 1 and shape 2 job systems for non-malleable and malleable tasks respectively as the mean arrival interval and laxity parameters are varied. The other parameters are kept fixed as follows: *number of processors*=16 and α=0.5. We find that for a large range of parameter values, the performance benefits from tunability *augment* those resulting from malleable task execution. This expected behavior is explained by the observation that tunability denotes flexibility in resource utilization both at the individual task level as well as at the global application level. In contrast, although malleability might allow a particular task to execute in a flexible fashion, it does not permit this flexibility to extend to the entire application. Thus, tunability complements malleability, exposing additional information about application resource requirements to the underlying resource management system.

Our results show that for both non-malleable and malleable task executions, tunability yields substantial performance advantages, with respect to both system utilization and throughput. These improvements are most pronounced in the mid-portion of each parameter value range: when the system is moderately loaded, when the deadlines are not extremely loose, when the number of processors available is not significantly larger than the concurrency degree of a parallel task, and when the job shapes are not too similar. These system characteristics describe the operating point of most practical parallel systems, attesting to the significant performance potential of application tunability.

7.5 Summary

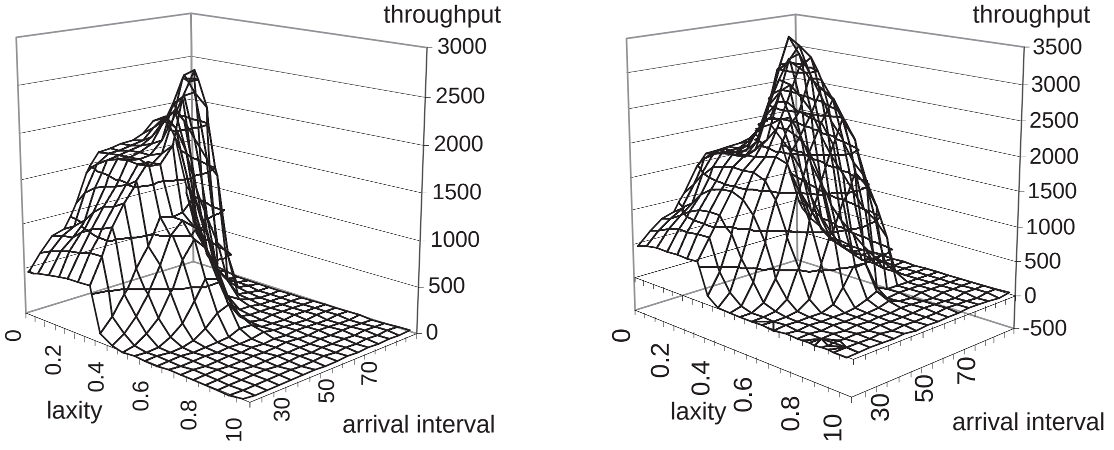
In this chapter, we studied the benefits of tunability by simulating the resource management in a parallel computing environment. The simulation shows that tunability



Benefit over shape 1

Benefit over shape 2

(a)



Benefit over shape 1

Benefit over shape 2

(b)

Figure 7.4: Performance impact of tunability in non-malleable model (a) and in malleable model (b), as job arrival interval and laxity are varied.

can greatly improve both system resource utilization and the number of jobs the system can finish. In a distributed environment, tunability can be more powerful due to more type of resources and dynamic changes of these resource conditions. A resource scheduler would have larger flexibility in matching application configurations to different resource conditions. In such a scenario, the benefits of application tunability also depend the amount of tradeoffs among various configurations and how much tradeoff end-users permit in the specification of QoS constraints. A larger tradeoff and more flexibility in fulfilling user preference makes tunability yield more benefits for both system and applications.

Chapter 8

Case Study: Active Visualization

This application is developed by the Active Visualization project at New York University. It has inherent flexibility in the way it can be executed. We take this application and expose this flexibility using our framework; and make it capable of adapting to different resource conditions. In this chapter, we will first describe the structure of this application and its tunability opportunities; show how to expose its various configurations using the tunability interface; generate profiles for its configurations using the virtual execution environment; and evaluate this framework by studying how it permits automatic adaptation to changes in resource availability.

8.1 Application Structure

The Active Visualization application [10, 11] is a client-server image visualization application for interactively viewing, at the client side, large images stored in the server. Figure 8.1 shows the overall structure of the client and server processes while Figure 8.2 shows a simplified version of the client-side source code. The application uses multi-

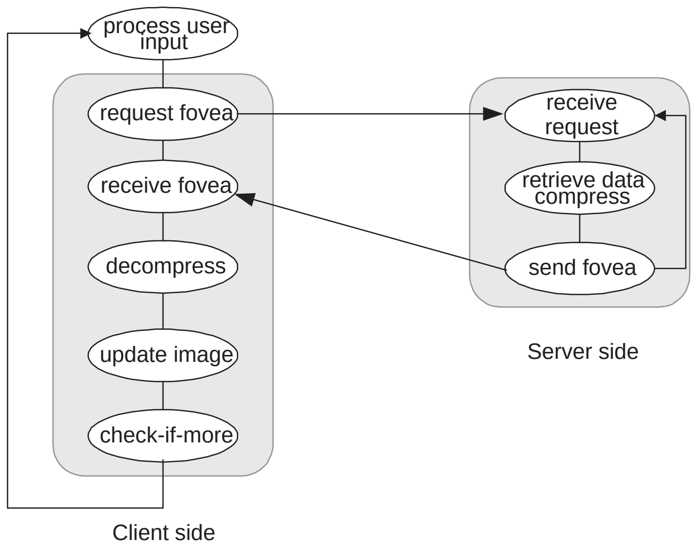


Figure 8.1: Structure of Active Visualization Application. The client requests a foveal region of the specific resolution level from the server, decompresses the data, and updates the local image on display.

resolution and progressive transmission techniques to improve performance. First, images (not compressed) are stored at the server as wavelet coefficients [10], enabling the construction of images at different levels of resolution. Second, the server employs progressive transmission to improve response time. It transmits an area of the image that corresponds to the user's fovea (i.e., focus of interest), starting from the coarsest resolution and progressing up to the user-preferred resolution.

Given a user request for a certain foveal region of which the center is indicated by variables `x` and `y` as in Figure 8.2 (corresponding to the position of the mouse) and the size by the variable `r`, the client sends a request to the server asking the corresponding region of a certain image resolution level as indicated by variable `l`. The server gets the request, compresses the data with some compression method configured by the client

```
establish_connection();
notify_server_compression_type(c);

do {
    check_for_user_interaction(&x, &y, &r, f);
    request_fovea(x, y, r, l);
    receive_fovea(&data);
    decompress(c, &data);
    update_image(x, y, r, l, data);
} while (check_if_more(r, l));

close_connection();
```

Figure 8.2: Structure of the original client side program of Active Visualization.

(as variable `c`), and sends the data back. The client receives the data, decompresses it accordingly, and updates the image. If the user has not moved the position of the mouse by then, the client automatically generates another request for a larger foveal region, expanding the fovea size by the value in variable f. This ensures eventual transmission of the entire image at the highest required resolution.

8.2 Tunability of Active Visualization

This application permits adaptation by associating different values with local variables such as preferred resolution level `l`, size increment of the foveal region `f`, and choice of the compression method `c`. The resultant executions trade off resource requirements for application performance: a low resolution implies less resource requirements; different compression methods require different CPU resources and achieves different compression ratio; and a smaller fovea size leads to a quicker response for getting the foveal region, but a longer time to transmit the whole image. In other words, the choice of resolution level trades off the image quality against the overall resource requirements; the choice of compression methods trades off CPU resource requirement against network resource requirement; the choice of fovea size trades off responsiveness against the total image transmission time. The setting of these control "knobs" defines different application configurations, which may be suitable for different execution scenarios. These knobs are naturally present in the original application program except the support of different compression methods. For compression methods, it turns out that original programmers experimented with several compression algorithms and finally decided the one that suits low network bandwidth, the execution scenario the application was designed for, without removing the rest of the algorithms.

For this application, the notion of tunability captures the fact that none of the execution scheme is optimal in all resource scenarios and that different schemes can trade off resource requirements and output quality. These tradeoffs enables the run-time selection of an appropriate execution scheme based on environmental conditions and user preference. In other words, tunability provides flexibility, for this application, in both the way application executes and the way users can express their preferences.

8.3 Application Annotation

Application tunability of Active Visualization is exported using the tunability interface. Although the application has both a server-side and a client-side component, we restrict our attention to the client side, treating the server side as a black-box whose behavior is entirely determined by the messages sent to it from the client.

Annotation for identifying configurations

The different kinds of configurations are specified using a control parameter construct and a component construct as in Figure 8.3 and Figure 8.4

Control parameters The annotation **<control_param>** in Figure 8.3 has three **<param>** elements, identifying local variables `l`, `f`, and `c` as control parameters with the corresponding name `level`, `fovea`, and `compress` respectively. Control parameters determine the configuration (i.e., execution path) taken by the application and the behavior along this path.

Tunable components As in Figure 8.4, the entire data transmission module is treated as a single component named `transmit{level}{fovea}{compress}`. The control parameters in the component name are evaluated as name-value pairs when the component construct is instantiated at run time, and serve as a handle for referring to a specific component configuration. Inside these components is the annotated source code, and the specification of tunability parameters including control parameters, resource parameters, and QoS metrics.

Annotation for monitoring and controlling execution

For the purpose of monitoring and control application execution, we expose two application instances (i.e., client and server) and resource parameters of the execution environment, declare the performance metrics (i.e., QoS metrics), specify how to evaluate them, and define a transition function invoked when switching between different

Tunability Parameter Declaration

```
 < control_param>
    < param type="int" externname="level" localname="l"    />
    < param type="int" externname="fovea" localname="f"    />
    < param type="int" externname="compress" localname="c" />
< /control_param>

< resource_param>
    < host name="client" address="localhost" local="yes"
          port="vSys.port+1"                             />
    < hostresc type="double" name="cpu" host="client"       />
    < hostresc type="double" name="net" host="client"       />
    < host name="server" addressvar="vSys.chosenServer"
          port="vSys.port+1"                             />
< /resource_param>

< QoS_param>
    < qos type="double" localname="qos_transmit"
          externname="transmit_time" dir="dec"            />
    < qos type="double" localname="qos_response"
          externname="response_time" dir="dec"            />
    < qos type="int" localname="qos_resolution"
          externname="resolution" dir = "inc"             />
< /QoS_param>
```

Figure 8.3: Declaration of tunability parameters using annotation, including control parameters, resource parameters, and QoS metrics.

Tunability Annotation of Active Visualization

```
< component name="transmit{level}{fovea}{compress}" >

        Tunability parameter declaration, e.g., control parameters, ...

< QoS_monitor>

    double qos_transmit = 0, qos_response = 0;

    int qos_resolution = l, nrounds = 0;

< /QoS_monitor>

establish_connection(); notify_server_compression_type(c); do {
    < QoS_monitor> double t0 = clock(); < /QoS_monitor>

    check_for_user_interaction(&x, &y, &r, f);

    request_fovea(x, y, r, l);

    receive_fovea(&data);

    decompress(c, &data);

    update_image(x, y, r, l, data);
    < QoS_monitor>

        double t1 = clock(); nrounds ++;

        qos_transmit = qos_transmit + (t1 - t0);

        qos_response = qos_transmit/nrounds;

        qos_resolution = l;

    < /QoS_monitor>

    < transition from="oldctl" to="newctl">

    if (oldctl.compress != newctl.compress)

        notify("server", "compress", (void *)newctl.compress);

    < /transition>

 } while (check_if_more(r, l));

< /component>
```

Figure 8.4: Exporting component tunability interface for the client slide of Active Visualization.

configurations.

Execution environment The execution environment (as **<resource_param>** in Figure 8.3) specifies two hosts (through **<host>**) on which the application executes, a `client` (running on `localhost`) and a `server` as well as their addresses and port numbers control messages should send to. The addresses and ports can be expressions valid at the point of annotation. The client host encapsulates CPU and network resources (specified by **<hostresc>**) of interest to the application. Here, resources on server host are not explicitly included, although it is also possible to monitor and adapt to the changes of the server-side conditions.

QoS metrics The **<QoS_metric>** construct in Figure 8.3 specifies (through **<qos>** elements) three metrics of interest, including the total data transmission time (i.e., `transmit_time`) that maps to local variable `qos_transmit`, average response time (i.e., `response_time`) that maps to local variable `qos_response`, and image resolution level (i.e., `resolution`) that maps to local variable `qos_resolution`. The performance level increases with a smaller `transmit_time`, `response_time` as indicated by `dir="dec"`, or a bigger `resolution` as indicated by `dir="inc"`.

Evaluation of QoS metrics As shown in Figure 8.4, updates to QoS metrics are handled by code segments contained within **<QoS_monitor>** constructs. The code inside these constructs are language-specific. Here we use C/C++ code declaring local variables that maps to QoS metrics and assign them proper values. For instance, wall clocks are taken in the beginning and the end of each round of transmission to get accumulated time as transmission time.

Transition functions Annotation in Figure 8.4 specifies one **<transition>** construct, containing application-specific actions for switching between configurations. It also presents point at which adaptation can happen inside a component. Here, the variable `newctl` represents the new configuration suggested by the run-time system while `oldctl` represents the old configuration. For this application, reconfiguration may require notifying the server when the compression method is to be changed. Here, `notify` is a function provides as an API that can be used in annotation for communicating between different application instances, and the run-time agents. After executing the transition function, the control parameters get the value of the `newctl`.

Interface for adaptation at binary level

Since we already know the control flow of Active Visualization and prototype and semantics of its function calls, we use function interception technique to turn the original binary into a tunable version. Although an automated implementation would require support for specifying which function to reinterpret. We decide to manually author the new functions for `check_for_user_interaction`, `decompress` and `check_if_more`. Inside these new functions, the original functions are invoked. These new functions present source code so that annotation constructs could be used to provide a similar tunability interface.

For instance, we redefined function `check_for_user_interaction` as in Figure 8.5 and add **<control_param>** and **<QoS_monitor>** constructs as in Figure 8.6 (other constructs are not shown). These constructs identifies `f` as a control parameter and starts the clock for measuring QoS metrics (e.g., transmission time). Similarly, the new function `check_if_more` is annotated with the another **<QoS_monitor>** con-

New `check_for_user_interaction` function

```
void check_for_user_interaction(int* px, int* py, int* pr, int f) {
    original_check_for_user_interaction function(px, py, pr, f);
}
```

Figure 8.5: New definition for the intercepted function

Annotation for the new `check_for_user_interaction` function

```
void check_for_user_interaction(int* px, int* py, int* pr, int f) {
    < control_param>
        < param type="int" externname="fovea" localname="f" />
    < /control_param>
    < QoS_monitor> t0 = clock(); < /QoS_monitor>

    original_check_for_user_interaction function(px, py, pr, f);
}
```

Figure 8.6: Annotation of intercepted functions.

struct and a **<transition>** construct to evaluate QoS metrics and triggers adaptation (if necessary) at the end of each data request-reply round.

8.4 Application Profile

A driver program repeatedly executes different configurations of the Active Visualization application in the virtual execution environment, obtaining a mapping from the control parameter values to the output qualities for a wide range of resource conditions. Figures 8.7 and 8.8 discusses a small subset of these measurements and the resulting mappings. The measurements reported here are obtained on a Pentium 450MHz machine and rely upon manual determination of appropriate resource settings for the virtual execution environment. The latter could be assisted by a sensitivity analysis tool that suggests the configuration and resource ranges requiring further profiling.

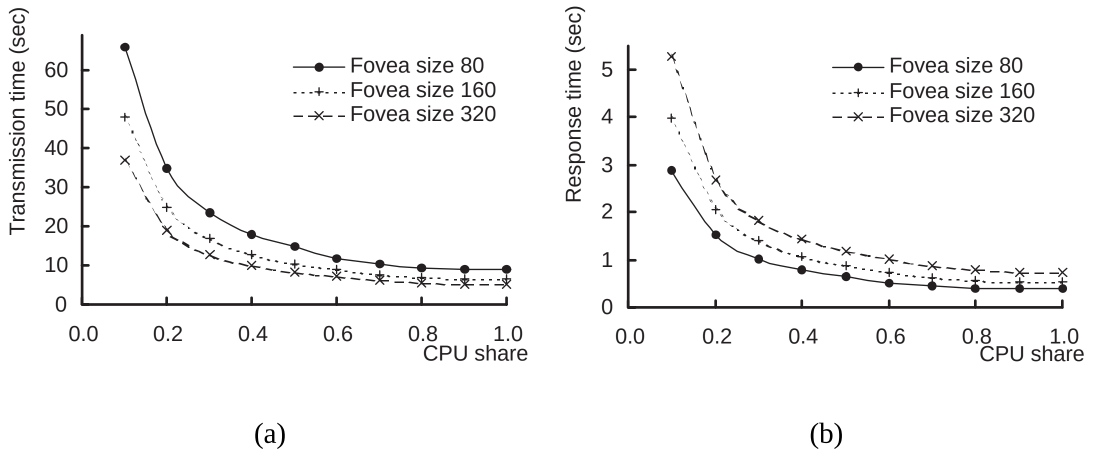
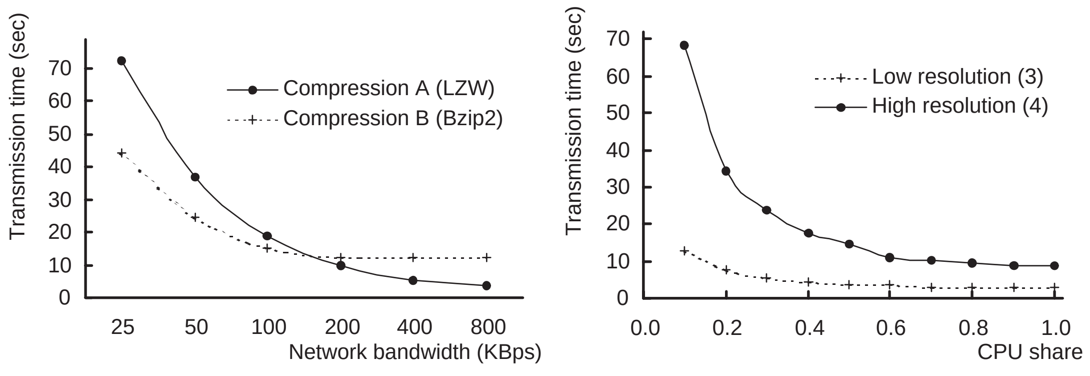


Figure 8.7: Image transmission time (a) and response time (b), for different fovea sizes as CPU share varies.

Figure 8.7 shows the image transmission time (i.e., `transmit_time`) and average response time (i.e., `response_time`) of user interactions for different fovea sizes as the client-side CPU share varies in the virtual execution environment (this corresponds to running the application on client machines with different CPU capabilities). In general, an increase in CPU resources reduces both transmission time and response time.



(a) (b)

Figure 8.8: Image transmission time, for (a) different compression methods as network bandwidth varies, and for (b) images of different resolutions as CPU share varies.

However, they show opposite trends (seen in the order of the curves) with the increase of fovea size: the larger the fovea size, the smaller the total transmission time, but the larger the response time.

Figure 8.8(a) shows image transmission time for different compression methods as the network bandwidth varies (keeping other resources such as CPU at a fixed level). The two curves in the figure correspond to two different compression methods in the application: compression B (Bzip2) trades off additional CPU resources to achieve a better compression ratio than compression A (LZW). The crossover between the two curves indicates that there exist resource conditions where one compression method should be preferred over another. Compression B outperforms compression A when the network bandwidth is low because less data is transmitted. However compression A performs better when the network bandwidth is high because the CPU becomes the bottleneck then.

Figure 8.8(b) shows the transmission time for images of different resolutions as the

CPU resource varies (keeping other resources at a fixed level). In general, additional CPU resources lead to a shorter transmission time. However, transmission time can also be lowered by degrading the image resolution.

Although we use exhaustive profiling to learn the behavior of all of the configurations, it is not necessary to store all the profiling result in the performance database. Only those configurations that outperform other configurations under some circumstances are recorded in the database, and for only those resource points that are necessary for a close interpolation. Since obtaining application behavior requires a large number of profilings , it is desirable to automate this process by providing tools that can pinpoint which resource conditions and configurations need further investigation.

A difficulty we experienced with the profiling stage is the abnormal execution of applications. When we run applications in an resource conditions they are not tested (or even not designed for), it is likely to experience abnormal executions such as crash due to the bugs in application program. We are currently not attempting to solve this problem.

8.5 Evaluation of the Framework

Because of the interactive nature of the application, the output quality metrics of most interest to the user are image resolution and timeliness, although their relative importance varies from situation to situation. To capture a wide range of usage scenarios, we describe three experiments below, demonstrating that our framework permits successful automatic adaptation to different patterns of changes in resource availability. Figure 8.9, Figure 8.10 and Figure 8.11 show the quality metrics achieved by the adaptable form of the application and contrast it with the non-adaptive versions. In each plot, the thick line represents the performance of the adaptive execution, which in each experiment switches from one configuration to another configuration. To better understand why this happens, we execute the application with the fixed configurations ever taken by the adaptive execution and plot as the two thinner lines in each figure.

The experiments emulate the client downloading ten images from the server. The experiments are conducted on two Pentium II 450MHz machines connected by 100 Mbps Ethernet. Here, we focus on adaptation to variations in CPU and network resources, and on only the client side of the application since the latter is more likely to be concerned with output quality metrics such as image resolution and response time. To test whether the application can adapt to run-time variations in resource conditions, we vary one of the resources (either CPU share or network bandwidth) after a fixed time in the experiment.

Experiment 1: Adapting compression method to network conditions Figure 8.9 shows the adaptation of the Active Visualization application in response to changes in the network bandwidth available between the server and the client. The user preference is to minimize image transmission time.

The network bandwidth is varied as follows: at the start of the experiment, the virtual execution environment provides a bandwidth of 500 KBps, which is changed to 50 KBps after 25 seconds. The resource scheduler responds to this pattern of resource availability as below:

- At startup, it configures the application to use compression method A (LZW). As Figure 8.8(a) shows, for a bandwidth of 500 KBps, compression method A (LZW) outperforms compression method B (Bzip2). This choice allows the application to download four images before the available bandwidth changes at time 25 seconds.

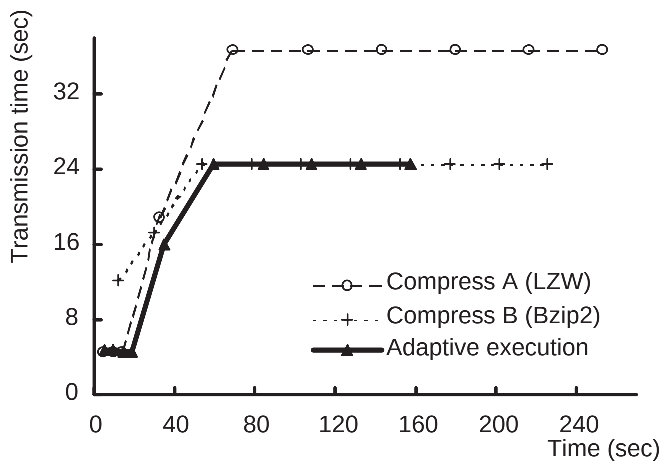


Figure 8.9: Adapting Active Visualization to changes in network conditions by switching compression methods.

- The change in bandwidth is detected by the application monitoring agent before the end of the fifth image transmission, which notifies the resource scheduler. The latter (based on the correlation in performance database) suggests the application to switch to compression method B (Bzip2), which the application-specific transition function realizes by sending the corresponding control message to the server (since compression is performed at the server side). As Figure 8.8(a) shows, compression method B yields better performance than compression method A when the bandwidth is 50 KBps. The switch takes effect in the middle of transmitting the fifth image, which takes 16 seconds to complete. Subsequent image transmissions use compression B and complete in 24 seconds apiece.

Experiment 2: Adapting image resolution to CPU conditions Figure 8.10 shows the application adaptation in response to changes in CPU conditions. In this case, the user preference requires that image transmission time not exceed 10 seconds and that image resolution be maximized. For simplicity, we constrain image resolution to be one of two levels (referred to hereafter as level 3 and level 4).

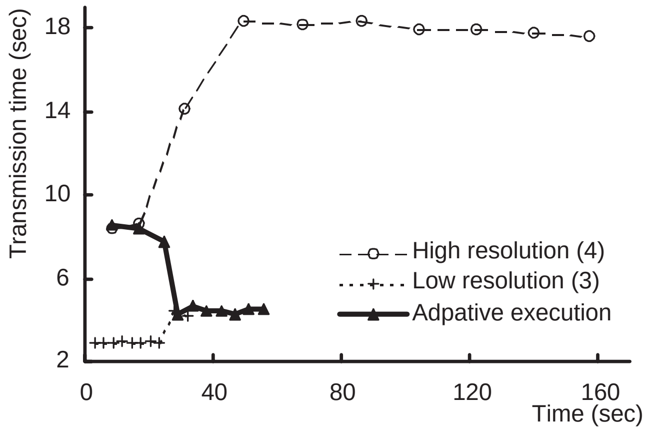


Figure 8.10: Adapting Active Visualization to changes in CPU conditions by degrading image resolution.

The CPU conditions at the client are varied as follows: at startup, the CPU share available to the client is set at 90%, which changes to 40% at time 30 seconds. The resource scheduler responds to this pattern of resource availability by starting off with a configuration that sets the image resolution to be level 4, but then degrades image quality to level 3 when resource availability drops. These choices are consistent with the performance profile shown in Figure 8.8(b). When the CPU share is 90%, resolution level 4 results in image transmission times within the user-requested range (i.e., less than 10 seconds). However, with a CPU share of 40%, image transmission times at this resolution level would violate user constraints. Degrading the resolution to level 3 permits each image to be transmitted in about 4 seconds, satisfying user requirements.

Experiment 3: Adapting fovea size to CPU conditions Figures 8.11 show application adaptation by changing fovea size in response to changes in CPU conditions. In this case, the user preference is to minimize image transmission time while keeping average response time of user interactions below 1 second.

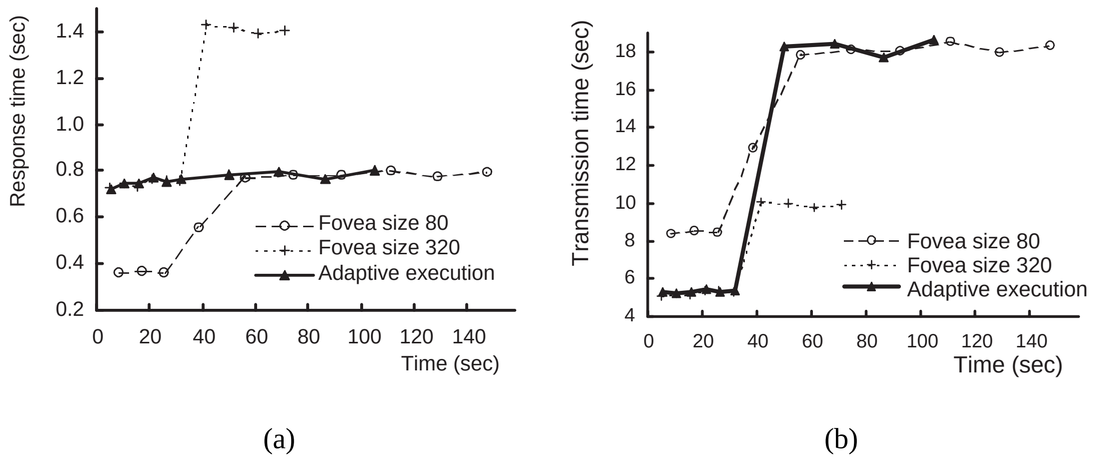


Figure 8.11: Adapting Active Visualization to changes in CPU conditions by changing fovea size.

The CPU conditions at the client are varied as follows: initially, the CPU share is set to 90%, but decreases to 40% at time 35 seconds. The two figures show response time (for retrieving the fovea) and transmission time (for retrieving the entire image) of executions under this resource availability pattern. The resource scheduler initially selects a fovea size of 320 (pixels), and switching down to a fovea size of 80 due the change in resource availability. These selections can be understood with the performance profiles shown in Figure 8.7. Initially, a fovea size of 320 satisfies the user preference for response times being below 1 second, while achieving the shortest image transmission time. However, when the amount of available CPU share drops, this configuration results in response times of about 1.4 seconds, which falls outside the user-requested range. Consequently, the scheduler switches to a fovea size of 80, ensuring response times of below 1 second for the remainder of the experiment. Compared with the configuration that satisfies this constraint (i.e., selecting fovea size 80), the adaptive execution achieves much shorter image transmission time.

We must note that although this experiment shows perfect adherence to the user preference, there are other issues that need to be addressed in a real execution scenario. In practice, application may experience constant irregular resource changes rather than shifting between stable levels, may have a very large performance database and complex user preferences. In this experiment, we take a simplified view, assuming stable resource levels and a simple version of performance database that gives the resource scheduler limited search space. This constraint yields a straightforward adaptation decision: Even though the CPU monitor takes several seconds to get a precise estimate after the resource change (as shown in Figure 6.2), an adaptation is triggered as soon any change is detected while the limited options in the performance database happen to satisfy the degraded stable CPU level. However, several issues may decide how adaptations happen in a real execution scenario, based on different policies including whether to react at the leading edge when any change is detected or at the trailing edge when a stable level is reached, and whether to choose a configuration that aggressively optimizes the objective function or one that conservatively keeps other constraints (e.g., the value of a QoS metric) well within prescribed bounds. The choice could affect the how responsive the application adapt and how well user preferences are satisfied.

8.6 Summary

Relying on the notion of application tunability, our framework enables automatic adaptation of the Active Visualization application. We first specify its control knobs using the tunability interface, identifying three control parameters: image resolution level, (increment of) fovea size, and compression method. The tunability interface allows enumeration of various configurations through different setting of control parameters, and measuring of their performance reflected in the value of QoS metrics. The virtual execution environment permits the control of diverse resource availability, therefore making it possible to find out which configuration is suitable for what execution scenarios. At run time, a monitoring agent detects the resource availability, providing information to the resource scheduler which could dynamically find the most appropriate execution scheme for the application. As a result, we observe a big improvement on both user satisfaction and application performance in an environment with dynamic resource conditions. For this application, tunability opportunity is convenient to express and it enables the framework to make appropriate adaptation decision automatically.

In the profiling and run-time adaptation stage of this application, we have been assuming the resource condition at the server side is fixed. In fact, our client is the only client accessing the server. However, the resource condition at the server side is very likely to change when multiple clients request images at the same time. For such a scenario, we could add additional parameters in execution environment representing the load of the server. This could be a single parameter (as an event) specifying the number of clients requesting service at the same time, or two parameters specifying the CPU share and network bandwidth available to the thread serving the client. As a result, monitoring agents would run on both the client side and the server side, the notification messages send to the steering agents (that export the parameters in the tunability interface), which in turn ask for the resource scheduler for a new configuration when resource condition changes. The new configuration is selected for both the client and the server, which set up their own control parameters to choose a different configuration (no action by the server if it exposes only environment parameters but no control parameters); therefore, achieving adaptation based also on the load of the server host.

Chapter 9

Case Study: Junction Detection

Junction Detection application is an image processing application developed by the vision research group at New York University. It detects a set of pixels in an image where color or intensity changes abruptly, serving as a core component of many image processing applications. We parallelize it to run on the Calypso system—a metacomputing environment that supports concurrent application on a cluster of workstations. Calypso applications have the flexibility in utilizing a variable number of machines. We take this application and expose its flexibility using our framework. In this chapter, we will first describe the structure of this application and its tunability opportunities; show how to expose its various configurations using tunability interface; generate profiles for its configurations; and evaluate this framework using experiments in which the characteristics of input images and the image arrival patterns are dynamically changed.

9.1 Application Structure

The parallel version of Junction Detection is built using Calypso system, using Calypso language constructs and linked with Calypso run-time library. In this section, we first introduce Calypso system, and then describe the structure of the Junction Detection application.

9.1.1 Calypso System

Calypso takes advantage of two execution techniques with strong theoretical foundations [19, 43]—*two-phase idempotent execution strategy*, and *eager scheduling*—to provide programmers with the view of a fault-free virtual shared memory environment, even when the underlying resources may incur faults and exhibit wide variations in processing speeds.

The Calypso programming system views computations as consisting of several parallel tasks inserted into a sequential program (see Figure 9.1). These parallel tasks running on *worker* machines are responsible for performing the computationally intensive work, while the sequential code (*Manager*) is responsible for the high-level control-flow and I/O. Within a parallel step, Calypso supports CREW (concurrent read, exclusive write) semantics to shared data structures, with updates visible only at the end of each parallel step. Additionally, the parallel tasks are idempotent, implying that a code segment can be executed multiple times (with possibly some partial executions), with exactly-once semantics. These multiple executions mask any faults in the underlying resources.

Calypso augments standard C++ with four keywords: `shared`, `parbegin`, `parend`, and `routine`. Globally shared variables are declared using the keyword `shared`.

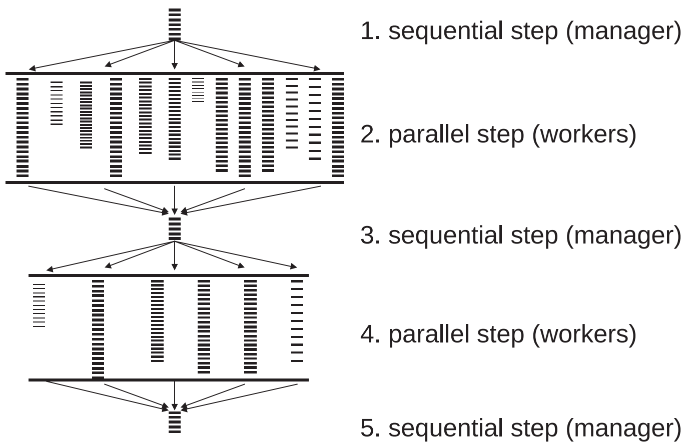

1. sequential step (manager)
2. parallel step (workers)
3. sequential step (manager)
4. parallel step (workers)
5. sequential step (manager)

Figure 9.1: A fragment of an evolving Calypso computation, with interleaving sequential and parallel steps.

`parbegin` and `parend` help delimit a parallel step consisting of a sequence of `routine` statements:

```
parbegin
  routine [int-exp](int width, int number)
     routine-body1
  routine [int-exp](int width, int number)
     routine-body2
parend;
```

The `routine` statements specify tasks within the parallel step: *routine-body1* and *routine-body2* are sequential C++ program fragments, *int-exp* specifies an integer expression indicating the number of copies of each routine to be created within the parallel step, and *width* and *number* are arguments provided to each task denoting, respectively, the number of tasks created and the sequence number of the specific task among these tasks. As shown in the code fragment above, each parallel step may consist of multiple `routine` statements. Concurrency exists both inside one routine and among multiple routines within the same parallel step. Calypso applications are able to utilize, in each parallel step, as many machines as the number of copies of all routines of the step.

Calypso system is implemented on both Windows NT and Linux operating systems. The version [18] on Windows NT allows allocating to applications only through a graphical user interface (GUI). The interactive feature of the interface presents a natural way to automatically control the application execution environment (in terms of the number of machines allotted).

9.1.2 Junction Detection Application

The Junction Detection [38] application detects distinguished sets of pixels in an image where the intensity or color changes abruptly, often serving as a precursor to shape construction and classification tasks. Our Junction Detection computation consists of three steps as in Figure 9.2. The first step samples a subset of the pixels in parallel in some granularity and performs a quick test to determine whether or not the tested pixel is of interest. A pixel is of interest if the difference among intensities/colors of its neighbor pixels is beyond a threshold. The second step draws a region of interest around a cluster of interesting pixels. The region is essentially a convex hull containing at least a certain number of interesting pixels in close proximity. Finally, the third step runs a computation intensive algorithm for every pixel in the regions of interest to detect all the junctions. Figure 9.3 illustrates a simplified version of the source code of this application.

The execution of Junction Detection application for processing an image involves a manager machine and a few number of worker machines. When multiple images need to be processed at the same time, multiple managers run on the same machine, with a

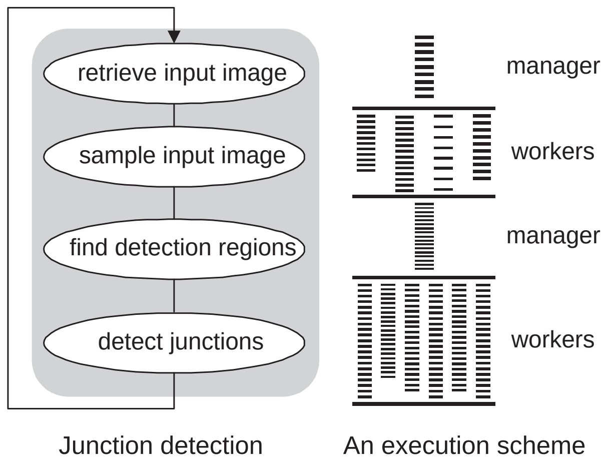


Figure 9.2: Structure of the parallel Junction Detection application and a possible processor-allocation scheme.

cluster of worker machines space-partitioned or time-shared among multiple executions depending on the decision of a resource scheduler.

9.2 Tunability of Junction Detection

The Junction Detection application is tunable in several senses: for example, the computation can trade off the amount of resource requirements between different computation steps: a larger allocation of more processors in the first step may be able to compensate a smaller allocation of in the third step to achieve the same level of performance. Here, however, we focuses on only the choice of two parameters: the sampling scheme (i.e., granularity) and the worker machine allocation scheme (i.e., the number of machines allocated to parallel tasks).

The purpose of sampling is to reduce the size of regions to be processed and therefore the amount of computation in the third step; however, at the cost of more compu-

```
// retrieve input image

int num_workers = 8;
int sample_granularity = 4;
bool first_time = true;
if (first_time) {calypso_spawnWorkers(num_workers); first_time = false;}

// step 1: sample input image
parbegin routine[num_tasks](int total_task, int task_index) {
    // sample pixels based "sample_granularity" and "task_index"
} parend; // step 2: find detection regions

// ...

// step 3: detection junctions in regions of interest
parbegin routine[num_tasks](int total_task, int task_index) {
    // detect junctions in regions allocated to "task_index"
} parend;
```

Figure 9.3: Original Calypso code of Junction Detection

tation for sampling itself in the first step. The exact amount of increment or decrement in computation depends on the algorithms used in these two steps, and thereby can be affected by the characteristics of the input image.

As a Calypso parallel application, a general parameter is the number of worker machines allocated to parallel steps. In general, with more machines allocated to the computation, the processing of the image is faster; however, the speedup may not be linear. In other words, with additional machines allocated, the application may not be able to utilize them as efficiently. In that case, it may be more efficient to give these machines to other computations.

The control parameters trade off resource utilization for application performance: allocating more worker machines to a single application may speed up its execution but degrade the overall system performance; and the arrival of images of different characteristics requires different sampling schemes to best utilize the system resources.

Identification of these control "knobs" permits adaptation by exploiting different number of worker machines for the parallel tasks and selecting appropriate sampling schemes for images with different numbers of interesting objects. Although these two parameters come from the flexibility of the Calypso system and the flexibility of the application structure respectively, both can be manipulated through the common control messages interface. The control messages in turn triggers different actions in the application: change of sampling scheme only involves setting of a variable whereas change of the machine allocation scheme requires control of the Calypso system, which is performed by simulating manual operation on Calypso-NT user interface (for Linux version of Calypso, this can be easily done through a Calypso API call).

For this application, tunability provides flexibility for optimizing performance based on external conditions, such as characteristics of input data and the overall resource requirement in the entire system.

9.3 Application Annotation

The Junction Detection application runs on both Windows NT and Linux with slight modification to fit different Calypso implementations. We focus on its Windows NT version where application is started and allocated resources through a GUI interface (which we can control by injecting transition functions to support adding and removing

Tunability Parameter Declaration

```
< control_param>
    < param type="int" externname="sample"
            localname="sample_granularity" />
    < param type="int" externname="worker"
            localname="num_workers" />
< /control_param>
< resource_param>
    < param type="int" name="image_class" />
    < param type="int" name="image_arrival_interval" />
< /resource_param>
< QoS_param>
    < qos type="double" externname="duration"
          localname="process_time" dir="dec"/>
< /QoS_param>
```

Figure 9.4: Declaration of tunability parameters for Junction Detection using annotation, including control parameters, resource parameters, and QoS metrics.

processors). The resources considered are the number of machines (i.e., processors) in a cluster of PCs. Although the resource description could be arbitrarily detailed including per-host parameters such as CPU load and network bandwidth, we restrict our attention to homogeneous hosts whose resource conditions can be abstracted in terms of just the number of available worker machines.

Tunability Annotation of Junction Detection

```
< component   name="junction{worker}{sample}" >

        Tunability parameter declaration, e.g., control parameters, ...

< QoS_monitor> double process_time = 0; double t0 = clock(); < /QoS_monitor>

int num_workers = 8;
int sample_granularity = 4;
bool first_time = true;
if (first_time) {calypso_spawnWorkers(num_workers); first_time=false;}
< transition from="oldctl" to="newctl">
if (oldctl.worker != newctl.worker) {// handshake with Calypso }
< /transition>
// step 1: sample input image
parbegin routine[num_tasks](int total_task, int task_index) {
    // sample pixels based "sample_granularity" and "task_index"
} parend;

// step 2: find detection regions
< transition from="oldctl" to="newctl">
if (oldctl.worker != newctl.worker) {// handshake with Calypso }
</transition>

// step 3: detection junctions in regions of interest
parbegin routine[num_tasks](int total_task, int task_index) {
    // detect junctions in regions allocated to "task_index"
} parend;
< QoS_monitor> process_time = clock() - t0; < /QoS_monitor>
< /component>
```

Figure 9.5: Exporting component tunability interface for Junction Detection, including various parameters (as in the thick box), and other constructs (in bold). The original source code is enclosed in thin boxes.

Annotation for identifying configurations

The entire image processing procedure can be viewed as one tunable component. The various configurations are different settings of this component. The tunability parameters and the annotation of the source code is illustrated in Figure 9.4 and Figure 9.5.

Control parameters The annotation **<control_param>** in Figure 9.4 identifies local variables `sample_granularity` and `num_workers` as control parameters, giving the name `sample` and `worker` respectively. These two parameters determine the sampling scheme and the worker machine allocation scheme that influence the behavior of this application.

Tunable components The processing of an image is treated as a single component named `junction{worker}{sample}` by a **<component>** construct as in Figure 9.5. Inside this component are the annotated source code, the specification of the tunability parameters including control parameters, resource parameters, and QoS metrics (i.e., the annotation code in Figure 9.4), and other annotation constructs such as **<transition>** and **<QoS_monitor>**.

Annotation for monitoring and controlling execution

The communication interface with various monitoring agents are defined as part of execution environment while the monitoring of application progress is specified using QoS metrics and their evaluation constructs (i.e., **<QoS_monitor>**). The transition functions specify the application-specific functions invoked when switching between different configurations.

Execution environment The execution environment, specified by **<resource_param>** in Figure 9.4 only considers the varying execution conditions, in this case, the characteristics of the input image and the image arrival interval. The characteristics of input data is listed as part of execution environment since we assume some agent (either inside or outside the application) can analyze the input and notify the application (in fact, its steering agent) of the classification result. It in general can be viewed as part of the context information of application execution. For an interactive application, both the input data and its classification can change dynamically. In this sense, the characteristics of input data is similar to the property of system resources.

QoS metrics The **<QoS_param>** construct in Figure 9.4 specifies a single performance metric of interest to this application, the duration for processing an image that maps to the local variable `process_time`. The performance level increases with a smaller duration. This metrics is measured by code segments contained within two `QoS_monitor` constructs in Figure 9.5.

Transition functions Configurations can change either in the sampling granularity or in the number of worker machines. The change of the latter needs to be accompanied by a handshake between the application and the underlying Calypso system. This handshake, expressed using the transition functions (i.e., **<QoS_param>** construct in Figure 9.5), results in the addition or removal of Calypso workers employed for executing the parallel tasks. A transition function is added before each parallel step, permitting assigning different number of machines to each of the parallel tasks.

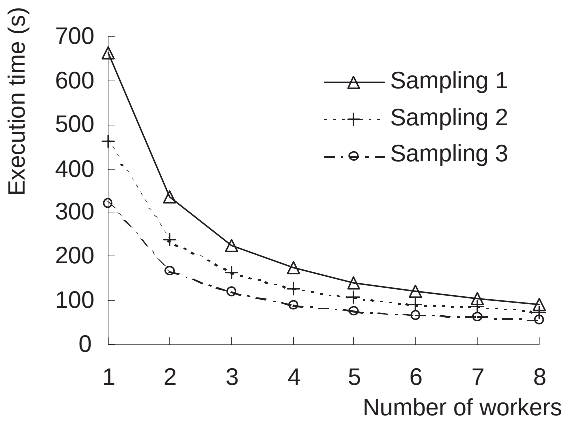
Interface for adaptation at binary level

Although we did not implement the tunability interface at binary level for enabling adaptation of Junction Detection application, the binary-level modification is straightforward and should be simpler than that of Active Visualization application because of the less number of control parameters and environment parameters. The only transition function is the handshaking with the underlying Calypso system, which is independent of a particular application and works for all Calypso applications. This means the handshaking code could be implemented inside a dynamic library (i.e., DLL) and invoked at proper points of intercepted functions (e.g., the function that implements `parbegin`).

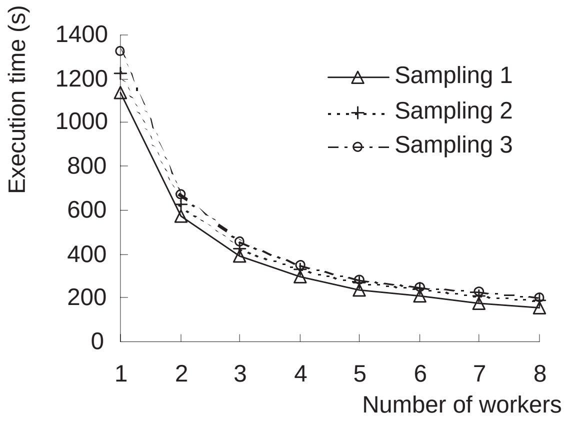
9.4 Application Profile

Similar to the Active Visualization application, a driver program repeatedly executes the Junction Detection application on a cluster of PCs to obtain a mapping between the control parameter values and the output qualities for a wide range of resource conditions, using images of different characteristics. For simplicity, we restrict our attention to execution environments where the only allowed change in resource conditions is the number of worker machines available to the application. This permits us to summarize the resource conditions in terms of a single parameter, the number of workers, assuming a homogeneous cluster with constant CPU speed and network bandwidth. To model input image characteristics, images are grouped into equivalence classes and representative subsets from these classes are profiled. The measurements reported here are obtained on a cluster of Pentium Pro 200 MHz worker machines connected by a 100 Mbps switched Ethernet.

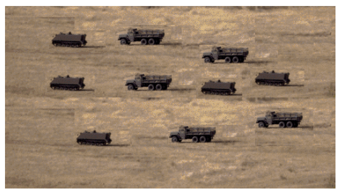
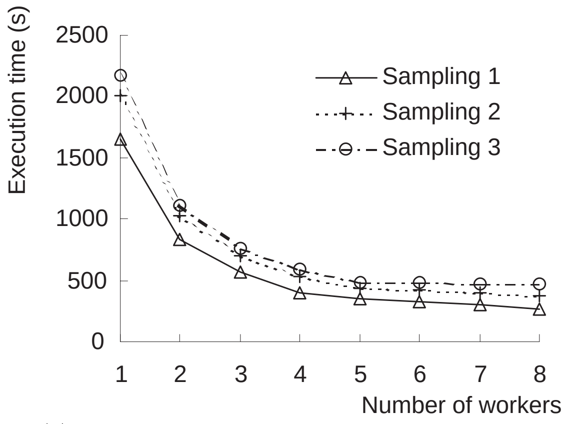
Figure 9.6 shows the execution time for images belonging to three representative



(a) image 1



(b) image 2

(c) image 3

Figure 9.6: Profiling images with different amount of artifacts (left) and the corresponding execution times of different sampling schemes as the number of machines allocated vary (right).

classes and for different sampling schemes as the number of worker machines is varied. The three sampling schemes (referred to as Sampling 1, Sampling 2, and Sampling 3 in Figure 9.6) correspond to the “sampling” control parameter being set to values of 4 pixels, 16 pixels, and 64 pixels respectively. The graphs show that the more artifacts an image has, the more time it takes to compute all of its junctions. However, different sampling schemes best suit images with different amounts of artifacts. Sampling 1 yields better performance for images with a large amount of artifacts (e.g., image 3), while Sampling 3 yields better performance for images with a small number of artifacts (e.g., image 1). In addition, as expected application execution time decreases as the number of worker machines increases; however, the speedup does not scale linearly attesting to the importance of having measured profiles.

9.5 Evaluation of the Framework

We use an NT implementation of the Calypso system, with a separate machine executing the manager code and manipulating the GUI interface. The user preferences of interest are to either minimize the execution time for detecting junctions within a single image, or to constrain this time to be within a deadline. The input to the application is a sequence of images belonging to different image classes, which arrive in the beginning or periodically with a specific interval. Over the entire run of the application, this period is assumed to be quasi-static; i.e., it changes only at a coarse granularity.

For adaptive execution, the external scheduler searches the performance database to find the best sampling scheme and number of workers to process a collection of images. The scheduler follows the standard interface described in Chapter 4: it takes an template that provides sufficient information about the tunability parameters and the format of the database. To make it capable of managing a cluster of machine for requests with timeliness constraints, we assume it knows the semantics of environment parameters, including the number of worker machines requested and the periodical nature of the image arrival pattern (in terms of the arrival interval). The external changes of concern are the arrivals of images of different classes and the variation of the arrival intervals. Both of the changes are assumed to be monitored by an external agent that informs the application through control messages sent to the steering agent. As described earlier, user preferences are satisfied by reconfiguring the application at run time; the granularity of the reconfiguration is the processing of a single image.

To capture typical usage scenarios of the Junction Detection application, we describe two experiments below that show adaptation of the application to changes in the characteristics and arrival patterns of input images. These experiments carefully isolate the possible adaptation behaviors of the application: the first experiment shows how the application can adapt to an image sequence comprising images belonging to different classes, when the number of workers available for processing a particular image is kept fixed. The second experiment demonstrates application adaptation in response to changes in the image arrival interval assuming that all images belong to the same class; here the application adapts by controlling the number of workers devoted to processing a single image. Figure 9.7 and Figure 9.8 show the per-image execution time achieved by the application for a sequence of images. The x-axis of the plots refers to the time when processing of a particular image is complete. As before, the solid line in each plot shows the performance of the adaptable form of the application and the other lines show performance achieved by the corresponding non-adaptable versions. Details of the image sequence and system conditions are provided below.

Experiment 1: Adapting sampling scheme to input classes Figure 9.7 shows adaptation of sampling schemes in response to arrival of different classes of input images, for a sequence of images whose classes are chosen randomly to be either type image 1 or type image 3 (see Figure 9.6). We assume that the number of worker machines allocated towards processing a single image is fixed to 4 and that all images arrive at the start of the experiment and are processed one by one. The user preference is to minimize the total execution time required for processing the image sequence.

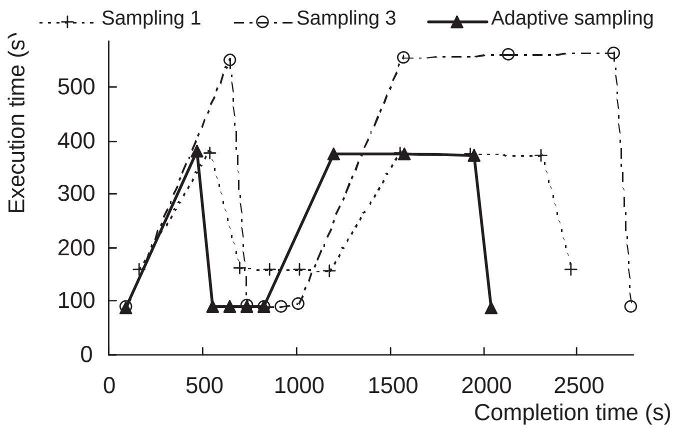


Figure 9.7: The Junction Detection application satisfies user preferences by adapting sampling schemes to different classes of input

The adaptable version of the application automatically chooses sampling scheme 3 for images of type 1 and sampling scheme 1 for images of type 3, when notified of the class of the arriving image. As a result, it finishes much earlier than non-adaptable versions of the application with fixed sampling schemes, with a 17% and 27% improvement over versions relying upon sampling scheme 1 and sampling scheme 3, respectively. The execution time shown here is slightly larger than in Figure 9.6 because of the startup overhead of Calypso GUI interface and the interference from the scheduler itself. We require the scheduler to consider the start-up cost in making dispatching decisions.

Experiment 2: Adapting worker allocation to image arrival patterns Figure 9.8 shows application adaptation by tuning the allocation of worker machines in response to variations in image arrival intervals. The user preference is to constrain the per-image processing time to be within a deadline of 120 seconds while trying to minimize the overall time required for processing the image sequence. The environment change is the image arrival pattern, which is initially set to one every 120 seconds, changes to one every 50 seconds after 200 seconds have elapsed in the experiment, and then reverts back to one every 120 seconds after 500 seconds. We use a total number of 8 worker machines and 1 manager machine, permitting the application to use any number of workers from 1 up to 8 for processing a particular image. The experiment uses images of type 1 as input and is set to stop after 800 seconds have elapsed.

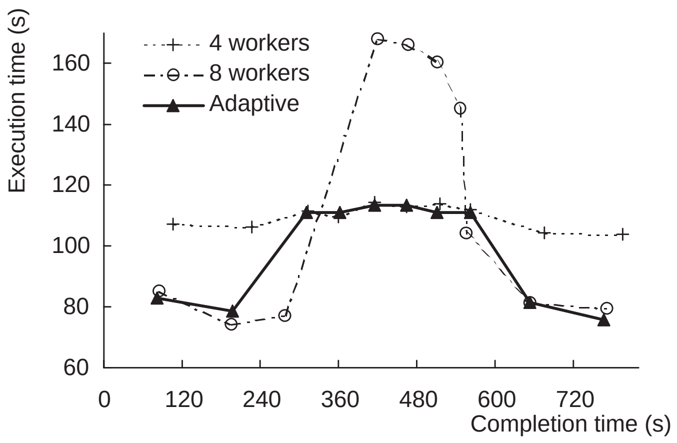


Figure 9.8: The Junction Detection application satisfies user preferences by adapting allocation of worker machines to variation in image arrival intervals.

Initially when the image arrival interval is 120 seconds, the application scheduler decides to give all the eight worker machines to each image because images do not overlap in their request for resources. When the image arrival interval changes to 50 seconds, the application gets notified by an external agent and triggers the scheduler to compute a new configuration. As a result, the scheduler decides to give each image 4 worker machines because two images may simultaneously need resources and must each finish within the specified deadline. In order to explain this scheduling decision, note from Figure 9.6(a) that the lowest execution time for processing images of class one is 100 seconds with 4 workers and 60 seconds with 8 workers (and that the scheduler takes into consideration extra time needed to start and control Calypso GUI interface). The initial allocation scheme would cause missing deadlines for a periodical arrival interval of 50 seconds. Similarly, the scheduler reverts back to allocating all machines to each image when the arrival interval changes back to 120 seconds. To compare how user preferences are satisfied with non-adaptive executions, we run the same image sequence again with two worker allocation schemes, which are fixed at using 8 machines or 4 machines respectively. The results show that with the first allocation scheme, some images miss their deadlines when the arrival interval is 50 seconds because at least two images (in the worst case four images) are time-sharing the worker machines at the same time, interfering each other's execution. For the second allocation scheme, all the images finish by the deadline (the slightly longer execution time in the middle of the curve is due to the sharing of the manager machine and network resources). However, the total execution time is 10% longer than with the adaptive execution.

9.6 Summary

The Junction Detection application exhibits its tunability in the sense that different execution schemes are best suited to different execution context. We first specify its control knobs using the tunability interface, identifying two control parameters: sampling scheme and machine allocation scheme. The tunability interface allows enumeration of various configurations through different setting of control parameters, and measuring of their performances as values of QoS metrics.

Since only resources are assumed to be homogeneous machines which can be represented simply by a number, we do not need to control their CPU and network availability using our virtual execution environment. The profiling is performed on a cluster of PCs, enumerating different configurations and varying parameters for the number of worker machines and the characteristics of input data. The profiling information plays a critical role for the scheduler to automatically configure the application to select an appropriate execution. As result, the framework automates the adaptive selection of sampling scheme and worker machine allocation scheme based on external conditions, such as characteristics of input data and overall requests for system resources. These adaptive executions greatly improve the performance of this application and user preferences. For this application, tunability allows natural expression of various configurations and enables the scheduler to make proper adaptation decisions.

Chapter 10

Concluding Remarks

This chapter summarizes this dissertation and discusses future work as well as the direction that its general thesis leads to.

10.1 Summary of This Dissertation

We observe that many applications naturally exhibit *tunability* opportunity, permitting selecting appropriate behavior from multiple configurations; however, no single configuration is always preferred for all execution scenarios. In this dissertation, we have described the notion of *application tunability* as flexibility in application execution schemes that trade off resource requirements and output quality among alternate configurations. These configurations may best match different execution conditions. This observation motivates automatic application adaptation taking advantage of tunability opportunity in applications. In this dissertation, we have developed an application-independent adaptation framework that simplifies the design of tunable applications, which proactively monitors the underlying environment and actively controls application execution to ensure a desired level of service.

To expose the alternate application configurations as well as the way to control and measure their execution, this framework supports an application-independent tunability interface, realizable either at the source code level using an XML-like annotation constructs or at the binary level using function interception combined with annotation. In each case, the preprocessor generates code for tunable applications based on the annotated program. To model the behavior of various application configurations, this framework builds a resource-constrained virtual execution environment using API interception and modern OS features allowing control of the amount of resources available to application execution. A driver program can profile these configurations by varying resource conditions on the virtual execution environment. To monitor the resource availability to application execution, this framework constructs run-time monitoring agents that proactively monitors resource conditions. To control application execution dynamically, this framework injects a steering agent into application that communicates with other run-time components and application instances using control messages. Combined with inline code generated from the tunability interface, the steering agent can set up alternate configurations. To make adaptation decisions based on execution environment, this framework has experimented with different resource schedulers that can make proper decision at run time to satisfy user preferences. Similar resource schedulers can be plugged into this framework by complying with its control message interface and exploiting application tunability parameters and performance databases.

We have evaluated the advantage of tunability and our framework using simulation of a parameterizable system and two large applications: an interactive distributed image visualization application and a parallel image processing application on Windows

NT platform. The simulation shows that application tunability can benefit both application execution and overall system resource utilization. The experiments with the applications show automatic adaptation to changes of execution environment characteristics such as network bandwidth or image arrival pattern by switching application configuration to satisfy user preferences for output quality and execution timeliness.

10.2 Conclusion

This dissertation has shown that the availability of *alternate application configurations* provides a powerful model for the construction and execution of adaptive applications. Exposing and constructing of application tunability can be achieved by exploiting source-level annotation, modeling application behavior on the virtual execution environment, allowing automatic selection of configurations at run time based on user preferences and execution conditions such as resource availability. Such applications can adhere to a standard interface so that configuration of applications, measuring of execution, and run-time adaptation can be automated; which in turn can automate (to a large extent) the development of tunable applications. Our framework presents an example for application structuring as well as for evaluating and selecting among the alternate application configurations.

This framework eliminates the need for adaptation decisions to be explicitly programmed into the application by relying on three components: (1) a *tunability interface,* which exposes adaptation choices in the form of alternate application configurations while encapsulating core application functionality; (2) a *virtual execution environment,* which emulates application execution under diverse resource availability enabling off-line collection of information about resulting behavior; and (3) *run-time adaptation* *agents*, which permit the monitoring and steering of application execution. Together, these components permit automatic run-time decisions on *when* to adapt by continuously monitoring resource conditions and application progress, and *how* to adapt by dynamically choosing an application configuration most appropriate for the prescribed user preference.

10.3 Future Work

In this project, we have made some implicit assumptions to simplify our work. The following list some of them that can be addressed in the future work,

1. *Negligible adaptation overhead*, we have assumed that application adaptation incurs negligible overhead. However, this may not be the case for all applications and allowed adaptations. Future work could eliminate this assumption by adding overhead functions to approximate application behavior when adaptation is triggered. Resource schedulers would take into consideration of this cost when suggesting the most appropriate configurations.

2. *Switching to any configuration*, we have assumed that applications at run time can switch from any configuration to any other configuration. Future work could eliminate this assumption by supporting specification of the constraints for switching between configurations. Resource schedulers can only select configurations to which switching is allowed from the current configuration.

3. *Adaptation only to significant changes*, we have implicitly assumed that adaptation only happens when the resource availability changes significantly and at the same time ignored the issue of adaptation agility. Future work could study how responsive applications should react to external changes. Frequent reactions may

not be appropriate in many cases.

4. *No need for synchronization*, we have assumed there is no need for synchronizing between different application instances when adaptation is triggered. Even when there is a need for agreeing on the compression method in the Active Visualization application, we ignored the problem by requesting the server to identify the compression method at the beginning of each of its messages. However, adaptation synchronization may be in general required for multiple application instances to agree on the new configuration and adapt in an specific order (or at the same time). Future work may need to address this issue.

5. *Simple adaptation policy*, we have assumed that it is sufficient to express user-specific policies in QoS constraints (as user preferences), which only allows specification of a range of the QoS metrics and optimization of a single QoS metric. For some applications, users may want to express their policies with more powerful tools, such as generic equations of QoS metrics. This would require a symbolic interpretation of the policies as well as corresponding resource schedulers. Future work could conduct further investigation in this direction.

Future work could also strengthen the current prototype and provide more support to some components of the framework, including,

1. *Sensitivity analysis tools*, in our current work, we use exhaustive profiling to obtain the behavior of various application configurations. Some of the effort yield meaningless results either because a particular configuration benefits application in no circumstances or the performance under a resource condition has already be covered by (the interpolation of) other profiling results. Future work could look into sensitivity analysis tools that determine which points in resource availability space should be more thoroughly profiled, and perform selective profiling.

2. *Analytical approximation of performance database*, in current framework, performance databases are specified in a discrete form to enable the construction of generic resource schedulers. Future work could analyze data in performance databases to obtain an analytical approximation. In addition, this approximation could be combined with run-time feedback (in terms of current values of QoS metrics) to provide proper information for making correct adaptation decisions at run time.

3. *Binary-level tunability interface,* in our current work, we have supported adding tunability to application binaries using code injection and function call interception techniques, in a prototype for the Active Visualization application. Future work could extend the tunability interface to specify which function calls to reinterpret and to allow associating specific actions with the new function implementation. Ideally, the binary-level approach could wrap the original application binary and build a COM-like component to expose a tunability interface so that shrink-wrapped applications could adapt to execution environment based on user preferences.

10.4 Perspective

The generic spirit of this dissertation lies in the recognition of multiple configurations of a same logical computation unit that are suitable for different execution scenarios and in the effort of automating the selection among them adaptively for improvement in application performance. Although this work was done on Windows NT platform with a few case studies, this spirit can be extended to many other applications and other platforms such as resource-constrained PDA devices. Applications could have different configurations for these heterogeneous platforms that can work together interchangeably. For instance, when an application moves from one device to another, it can continue from where it was, but more importantly, can automatically select another configuration that best utilizes the new environment. It is even possible to have configurations that works across devices, having the computation at one platform and display or input at another platform.

For applications running on resource-constrained devices where computation and memory capability is relatively small and supporting wireless communication exhibits a diverse variance in bandwidth, it is desirable that these applications have a resource-efficient configuration when executing isolated on the device, and another capability-enhanced configuration that goes out to find necessary components or services on the network to combine into itself when connected with outside world. These multiple configurations may even include binary code for different platforms within the same container, for instance, a form for running on PalmOS, another form for running on Linux, and a third form for running on Windows NT/2000. This container can be stored on a server (or itself can be a server) that allows code downloading to various platforms. A Palm application can just download the appropriate configuration, install and run it on the fly without having it stored permanently on the device; or even more aggressively, download the appropriate configurations of various components and build up the application just before execution.

This leads to another direction for extending the current work—*on the fly construction* of application configurations, which could seek support from the existing *component model* such as the Microsoft COM and the Sun JavaBeans, where components export standard interfaces and reflection mechanisms. Tunability interfaces could be implemented as part of what gets exported from a component. The reflection mechanism of the component model could be extended to support dynamic construction of a specific configuration on the fly. This configuration, in its compact form, may only have guidelines as to what components it needs and how to combine them together. It would be interesting if we can dynamically inject tunability opportunity (as well as the run-time agents) inside an existing component to expand its exported interfaces and turn it into a tunable component so that it can reflect on its different configurations, monitor its environment, and adapt itself to different execution contexts.

Although in the case studies included in this work, adaptation only happens at application end-points, some application instances might reside inside the connecting network between the end-points. The selection and combination of these components into applications can themselves be adapted to dynamic environment conditions. For instance, the output quality could be dynamically improved if the system finds more powerful components to substitute for the existing ones when realizing it just encounters a sub-network with more resources and services.

In summary, it is our belief that, with the current trends of increasingly heterogeneous resource-constrained devices, downloadable code, networking computing, and component model, application alternate configurations and dynamic construction and selection of these configurations would increase the satisfaction of user preferences by presenting more choices and appropriately adapting their behavior to the execution environment. Many projects are heading toward this direction. We believe that in the near future flexible application structuring and adaptation would occurs anywhere from inside network to application end-points, providing more powerful and satisfactory services to end users.

Bibliography

[1] R. Balzer and N. Goldman. Mediating connectors. In *Proc. 1999 ICDCS Workshop on Electronic Commerce and Web-based Applications*, Jun. 1999.

[2] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of 3rd USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Feb. 1999.

[3] A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proc. 4th IEEE Intl. Symp. on High Performance Distributed Computing*, 1995.

[4] F. Berman and R. Wolski. Scheduling from the perspective of the application. In *Proc. 5th IEEE Intl. Symp. on High Performance Distributed Computing*, 1996.

[5] T. Berners-Lee and D. Connolly. Proposed standard HyperText Markup Language Specification — 2.0. RFC 1866, Nov. 1995.

[6] D. Boling. *Programming Windows CE*. Microsoft Press, 1998. ISBN 1572318562.

[7] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse operating system:

Providing quality of service via reservation domains. In *Proc. of USENIX 1998 Annual Technical Conference*, Jun. 1998.

[8] R. Buff and A. Goldberg. Interactions between delayed Acks and Nagle's algorithm in HTTP and HTTPS: Problems and solutions. http://www.cs.nyu.edu/cs/faculty/artg/.

[9] K. Chandy and C. Kesselman. A description of CC++. Technical Report CS-92-01, California Institute of Technology, 1992.

[10] E. Chang and C. Yap. A wavelet approach to foveating images. In *Proc. 13th ACM Symp. on Computational Geometry*, 1997.

[11] E. Chang, C. Yap, and T. Yen. Realtime visualization of large images over a thinwire. In *IEEE Visualization '97*, 1997.

[12] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level resource-constrained sandboxing. In *Proc. 4th USENIX Windows System Symposium*, Aug. 2000.

[13] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *Proc. 9th IEEE Intl. Symp. on High Performance Distributed Computing*, 2000.

[14] F. Chang and V. Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Journal of Cluster Computing*, 2001.

[15] F. Chang, V. Karamcheti, and Z. Kedem. Exploiting application tunability for efficient, predictable parallel resource management. In *Proc. 13th Intl. Parallel Processing Symposium*, 1999.

[16] F. Chang, V. Karamcheti, and Z. Kedem. Exploiting application tunability for efficient, predictable resource management in distributed and parallel system. *Journal of Parallel and Distributed Computing*, 2000.

[17] S. Chatterjee. Dynamic application structuring on heterogeneous, distributed systems. In *Proc. IPPS/SPDP'99 Workshop on Parallel and Distributed Real-Time Systems*, 1999.

[18] P. Dasgupta. Parallel processing with Windows NT networks. In *Proc. USENIX Windows NT Workshop*, Aug. 1997.

[19] P. Dasgupta, Z. Kedem, and M. Rabin. Parallel processing on networks of workstations: Fault-tolerant high performance approach. In *Proc. 15th IEEE Intl. Conf. on Distributed Computing Systems*, 1995.

[20] M. Dertouzos. Control robotics: The procedural control of physical processes. In *Proc. of the IFIP Congress*, Aug. 1974.

[21] R. Englander. *Developing Java Beans*. O'Reilly & Associates, 1997. ISBN 1565922891.

[22] eTesting Labs. Winbench 99. http://www.zdnet.com/zdbop/winbench/winbench.html.

[23] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, 1999.

[24] W. Feng and P. Tinnakornsrisuphap. The failure of TCP in high-performance computational grids. In *Proc. Intl. Conf. for High Performance Computing and Communications (SC2000)*, Nov. 2000.

[25] D. Fields and M. Kolb. *Web Development with Java Server Pages*. Manning Publications Company, 2000. ISBN 1884777996.

[26] G. Geist and V. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4), 1992.

[27] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proc. of 6th USENIX Security Symposium*, Jul. 1996.

[28] S Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The Ninja architecture for robust internet-scale systems and services. *Special Issue of Computer Networks on Pervasive Computing*, 2001.

[29] L. Griffith. Precision NT event timing. *Windows Developer's Journal*, Jul. 1998.

[30] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994. ISBN 0262571048.

[31] T. Gross, P. Steenkiste, and J. Subhlok. Adaptive distributed applications on heterogeneous networks. In *Proc. 8th Heterogeneous Computing Workshop (HCW '99)*, Apr. 1999.

[32] J. Haartsen. Bluetooth—the universal radio interface for ad hoc, wireless connectivity. *Ericsson Review*, 3, 1998.

[33] J. Hollingsworth and P. Keleher. Prediction and adaptation in Active Harmony. In *Proc. 7th Intl. Symp. on High Performance Distributed Computing*, Apr. 1998.

[34] D. Hull, W. Feng, and J. Liu. Operating system support for imprecise computation. In *Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities*, Nov. 1996.

[35] D. Hull and J. Liu. ICS: A system for imprecise computations. In *Proc. AIAA Computing in Aerospace 9*, Oct. 1993.

[36] D. Hull, A. Shankar, K. Nahrstedt, and J. Liu. An end-to-end QoS model and management architecture. In *Proc. IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, Dec. 1997.

[37] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. 3rd USENIX Windows NT Symposium*, Jul. 1999.

[38] H. Ishikawa and D. Geiger. Segmentation by grouping junctions. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 1998.

[39] J. Itoh, R. Lea, and Y. Yokote. Using meta-objects to support optimisation in the Apertos operating system. In *Proc. USENIX Conf. on Object-Oriented Technologies (COOTS'95)*, Jun. 1995.

[40] M. Jones and J. Regehr. CPU reservations and time constraints: Implementation experience on Windows NT. In *Proc. 3rd USENIX Windows NT Symposium*, Jul. 1999.

[41] M. Jones and J. Regehr. CPU reservations and time constraints: Implementation experience on windows NT. In *Proc. of 3rd USENIX Windows NT Symposium*, Jul. 1999.

[42] J.Pritchard. *COM and CORBA Side by Side: Architectures, Strategies, and Implementations*. Addison-Wesley Pub Co, 1999. ISBN 0201379457.

[43] Z. Kedem, K. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proc. 22nd ACM Symp. on Theory of Computing*, 1990.

[44] P. Keleher, J. Hollingsworth, and D. Perkovic. Exploiting application alternatives. In *Proc. 19th Intl. Conf. on Distributed Computing Systems*, Jun. 1999.

[45] C. Koelbel, D. Loveman, R. Shreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. O'Reilly & Associates, 1997. ISBN 1565922891.

[46] C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek. On Quality of Service optimization with discrete QoS options. In *Proc. IEEE Real-time Technology and Applications Symposium*, Jun. 1999.

[47] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd Ed)*. Addison-Wesley Pub Co, 1999. ISBN 0201432943.

[48] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Proc. 7th IEEE Symp. on High-Performance Distributed Computing*, Jul. 1998.

[49] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proc. IEEE Intl. Conf. on Multimedia Computing and Systems*, May 1994.

[50] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, Jun. 2000.

[51] B.D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in Odyssey. *Mobile Networks and Applications*, 4, 1999.

[52] P. Chandra et al. Darwin: Customizable resource management for value-added network services. In *Proc. 6th IEEE Intl. Conf. on Network Protocols*, 1998.

[53] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using QDL to specify QoS aware distributed (QuO) application configuration. In *Proc. 3rd IEEE Intl. Symp. on Object-Oriented Real-time Distributed Computing (ISORC 2000)*, Mar. 2000.

[54] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proc. of SPIE/ACM Conf. on Multimedia Computing and Networking*, Jan. 1999.

[55] J. Richter. *Advanced Windows*. Microsoft Press, 1996. ISBN 1572315482.

[56] J. Richter. Microsoft .NET framework delivers the platform for an integrated, service-oriented web. *MSDN magazine*, Sep. 2000.

[57] M. Shankar, M. DeMiguel, and J. Liu. An end-to-end QoS management architecture. In *Proc. Real-Time Applications Symposium*, Jun. 1999.

[58] N. Spring and R. Wolski. Application level scheduling of gene sequence comparison on metacomputers. In *Proc. 12th ACM Intl. Conf. on Supercomputing*, 1998.

[59] Sun Microsystems, Inc. KVM white paper. May 2000. http://java.sun.com/products/cldc/wp/KVMwp.pdf.

[60] T. Thai. *Learning DCOM*. O'Relly & Associates, Inc., 1999. ISBN 1565925815.

[61] VMware, Inc. What is VMware? `http://www.vmware.com/whatisvmware.html`.

[62] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *14th ACM Symp. on Operating Systems Principles*, 1993.

[63] A. Williams. Scripting applets pays off. *Web techniques*, May 2000.

[64] J. Zinky, D. Bakken, and R. Schantz. Architectural support for Quality of Service for CORBA objects. *Theory and Practice of Object Systems*, Jan 1997.